UNIVERSIDAD CARLOS III DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE INFORMÁTICA

GRUPO DE ARQUITECTURA DE COMPUTADORES, COMUNICACIONES Y SISTEMAS

Implementación y análisis comparativo del rendimiento y eficiencia energética de un algoritmo de descompresión sobre FPGAs

PROYECTO FIN DE CARRERA

INGENIERÍA TÉCNICA INDUSTRIAL: ELECTRÓNICA INDUSTRIAL

Autor: Daniel Sánchez Sancho-Sopranis

Tutores: David Expósito Singh

Óscar Pérez Alonso

Mayo 2.012

Proyecto Fin de Carrera

TÍTULO: Implementación y análisis comparativo del rendimiento y eficiencia energética de un algoritmo de descompresión sobre FPGAs

AUTOR: Daniel Sánchez Sancho-Sopranis

TUTORES: David Expósito Singh
Óscar Pérez Alonso

La defensa del presente Proyecto Fin de Carrera se realizó el día 7 de Mayo de 2.012, siendo calificada por el siguiente tribunal:

PRESIDENTE: Ricardo Vergaz Benito

SECRETARIO: Daniel Higuero Alonso-Mardones

VOCAL: Soledad Escolar Díaz

Que acuerda otorgarle la calificación de:

PRESIDENTE:

SECRETARIO:

VOCAL:

AGRADECIMIENTOS

En primer lugar quiero dar las gracias a mis dos tutores, David y Óscar, ya que sin ellos este proyecto no se había llevado a cabo, en todo momento han puesto mucho interés en el proyecto y en todos los casos que he requerido ayuda ellos han estado allí para intentar solucionar mis problemas. Por todo ello han sido dos tutores muy buenos. También agradecer al responsable del grupo de investigación arcos, Jesús Carretero, sus visitas diarias interesándose por el tema en todo momento.

Como no podría ser menos quiero dar las gracias a mi familia por poner a mi disposición todos los recursos necesarios para que haya podido estudiar esta carrera, por sus consejos y experiencia y por el apoyo que me han dado en los momentos más difíciles de la carrera. Por otro lado también dar las gracias a mi novia, Irene, por su constante apoyo en los momentos de desesperación ya que siempre ha estado a mi lado y sin ella no podría haber terminado los estudios. Gracias a ellos por todo el cariño y ayuda que he recibido.

También quiero agradecer a todos mis compañeros de la carrera, con los que he compartido esos días, semanas y meses interminables con clases, prácticas y exámenes ya que sin ellos no podría haber aguantado esas jornadas de doce horas en la universidad. También por toda la ayuda que nos hemos dado mutuamente tanto a la hora de realizar las prácticas y estudiar para los exámenes ya que sin ese humor y bromas constantes para levantar el ánimo todo esto habría sido mucho más duro.

Y antes de llegar a la universidad ha habido otro largo camino de estudios que visto ahora no fueron tan duros (aunque no lo parecía en el momento) en el que he compartido grandes momentos con mis amigos de toda la vida, en el colegio e instituto. He crecido junto a ellos y parte de como soy es gracias a ellos y aunque no sea de forma directa pero también han contribuido a que pueda sacar esta titulación.

Y por último lugar agradecer a todos los buenos profesores que he tenido, ya que a parte de la dificultad de la asignatura el profesor cuenta mucho, y me gustaría agradecer a los profesores que han conseguido que me gusten asignaturas que a priori eran difíciles y gracias a eso me han animado a continuar con los estudios.

A todos ellos **muchas gracias** por influir en mí positivamente y conseguir de una forma u otra que este largo camino llegue a su fin.

RESUMEN

Este parte como trabajo futuro de otro proyecto en el que se implementó un algoritmo software de compresión de datos realizando un diseño teórico del mismo en lenguaje VHDL que se implementó en una FPGA. La idea inicial es implementar el algoritmo software de descompresión en una plataforma hardware mediante el lenguaje VHDL y realizar una evaluación del mismo sobre distintas plataformas. El descompresor consta de varios campos de tamaño totalmente configurable. Este proyecto forma parte del departamento de informática de la Universidad Carlos III de Madrid.

El proyecto ha tenido distintas fases, la primera de ella es la implementación del algoritmo de descompresión en lenguaje VHDL, para ello ha utilizado el entorno de desarrollo Quartus II del fabricante Altera, ya que permite una rápida compilación y una simulación sencilla para verificar el correcto funcionamiento del diseño. Una vez el funcionamiento teórico es correcto se ha procedido a su implementación en la FPGA Nexys 2 de Digilent aunque posteriormente se han realizado pruebas con tres FPGAs distintas a esta, en este caso ha sido necesario el uso del entorno de desarrollo Xilinx Platform Studio (XPS) de Xilinx.

Las FPGAs utilizadas son la Nexys 2, Nexys 3, Atlys (todas ellas de Digilent) y la Spartan 3A Starter Kit de Xilinx. Todas ellas constan de un microprocesador Microblaze que se ha configurado para que se encargue de la transmisión de datos al diseño en VHDL. El diseño en VHDL ha sido implementado en distintos coprocesadores, para que su funcionamiento sea totalmente independiente y modular. La conexión entre Microblaze y los distintos coprocesadores ha sido realizada mediante buses de comunicación FSL.

Una vez se ha comprobado el correcto funcionamiento en FPGA se ha implementado en un mismo diseño el algoritmo de compresión y descompresión, así es posible evaluar si el algoritmo y ambos diseños son correctos. La comprobación ha sido satisfactoria por lo que se ha procedido a realizar pruebas de velocidad del diseño, consumo energético, comportamiento en distintas placas y comparación del coste hardware en cada FPGA tales como biestables, DCMs, LUTs y otros recursos.

En último lugar se ha realizado un tutorial para ampliar los conocimientos del manejo del programa XPS, para permitir la comunicación mediante varios buses FSL desde Microblaze, lo que facilitará la implementación de futuros diseños.

Palabras clave: Xilinx Platform Studio, FPGA, coprocesador, Microblaze, FSL, descompresión, Quartus II, algoritmo.

ABSTRACT

This project is an extension of an earlier project which implemented a compression data algorithm. The former one includes a theoretical design in VHDL language of the compression module and a practical implementation in an FPGA. The general idea of this project is to extend the previous project by implementing and testing a decompression algorithm in a hardware device using VHDL. The selected decompressor consists of several configurable modules; each one is responsible of compressing a particular field of the data. This project was developed in the department of computer science at the University Carlos III of Madrid.

This project has different developing phases. The first one is the implementation of the decompression algorithm in VHDL language. We have used Quartus II framework from Altera, which performs fast hardware compilations and design simulations that allows to verify the correct design operation. In a second phase, we carried out a design implementation and testing in a FPGA Nexys 2 from Digilent. Then, this procedure was extended to other three different devices: Nexys 3, Atlys and Spartan 3A Starter Kit Xilinx. In these cases it has been necessary to use the Xilinx Platform Studio (XPS) framework from Xilinx.

In overall, four different FPGAs were evaluated: Nexys 2, Nexys 3, Atlys (all three from Digilent) and Spartan 3A Starter Kit Xilinx. All of them consist of a Microblaze microprocessor that is configured to communicate with the co-processors (compressor/decompressor modules). The VHDL design has been implemented in these co-processors, so that its operation is totally independent and modular. The connection between Microblaze and the co-processors has been performed using FSL communication buses.

Once we have verified the correct operation on FPGAs, we evaluated the compression and decompression algorithms, so it is possible to evaluate whether the algorithm and both designs are properly working in each platform. Then we tested our design for different platform speeds, measuring the power consumption, the performance and the hardware costs: number of biestables, DCMs, LUTs, etc.

Finally, this project includes a tutorial that extends the use of the XPS program. We introduce a methodology for performing communications through several FSL buses from Microblaze and coprocessors, which will simplify the design and implementation in future designs.

Keywords: Xilinx Platform Studio, FPGA co-processor, MicroBlaze

ÍNDICE

1. Introducción ... 8
1.1 Descripción del problema ... 8
1.2. Motivación ... 9
1.3. Objetivos ... 10
2. Estado de la cuestión ... 11
2.1. Introducción de FPGAs ... 11
2.1.1. FPGA Nexys 2 ... 13
2.1.2. FPGA Spartan 3 ... 14
2.1.3. FPGA Nexys 3 ... 15
2.1.4. FPGA Atlys ... 16
2.2. Introducción a la compresión de datos ... 17
3. Algoritmo de descompresión ... 19
3.1. Descripción del algoritmo de descompresión ... 19
3.1.1. Descripción del algoritmo de descompresión simple ... 19
3.1.2. Descripción del algoritmo de descompresión doble ... 21
3.2. Ejemplos de funcionamiento ... 23
3.2.1. Ejemplo funcionamiento descompresión simple ... 23
3.2.2. Ejemplo funcionamiento descompresión doble ... 24
4. Diseño hardware ... 28
4.1. Arquitecturas diseñadas ... 28
4.1.1. Arquitectura de descompresión simple ... 28
4.1.2. Arquitectura de descompresión doble ... 31
4.2. Descripción de las arquitecturas en alto nivel ... 34
4.2.1. Visión global del algoritmo ... 34
4.2.2. Descompresión simple ... 36
4.2.3. Descompresión doble ... 38
4.3. Descripción detallada de los algoritmos hardware ... 39
4.4. Definición del interface software ... 61
4.5. Integración de la arquitectura completa ... 65
4.5.1. Descripción de la arquitectura en alto nivel ... 65
4.5.2. Descripción interface Software ... 67
5. Estudio del rendimiento ... 69
5.1. Validación de la arquitectura ... 69

5.1.1. Quartus II 69
5.2. Análisis de velocidad de la arquitectura de descompresión en distintas FPGAs 72
5.2.1. FPGA Nexys 2 73
5.2.2. FPGA Spartan 3 78
5.3. Análisis de consumo de energía de la arquitectura de descompresión en distintas FPGAs 82
5.3.1. FPGA Nexys 2 83
5.3.2. FPGA Spartan 3 86
5.4. Análisis de velocidad de la arquitectura completa en distintas FPGAs 89
5.4.1. FPGA Nexys 2 89
5.4.2. FPGA Nexys 3 91
5.4.3. FPGA Atlys 93
5.5. Análisis de consumo de energía de la arquitectura completa en distintas FPGAs 95
5.5.1. FPGA Nexys 2 95
5.5.2. FPGA Nexys 3 96
5.5.3. FPGA Atlys 97
5.6. Análisis de coste hardware de los diseños en distintas FPGAs 101
5.6.1. Análisis coste hardware arquitectura de descompresión en distintas FPGAs 101
5.6.2. Análisis coste hardware arquitectura completa en distintas FPGAs 103
5.7. Comparación entre arquitectura en VHDL, y en C en Microblaze y CPU Intel Core I7 105
6. Conclusiones y trabajos futuros 110
6.1 Conclusiones 110
6.2. Trabajos futuros 112
7. Presupuesto 113
Apéndice A. Pasos en la implementación de un proyecto en una FPGA con varios FSL 115
8. Bibliografía 122

LISTA DE FIGURAS

Figura 1. Placa Nexys 2. 13
Figura 2. Placa Spartan-3A Starter Kit. 14
Figura 3. Placa Nexys 3. 15
Figura 4. Placa Atlys. 16
Figura 5. Esquema del algoritmo de compresión simple. 19
Figura 6. Esquema del algoritmo de descompresión simple. 20
Figura 7. Esquema del algoritmo de compresión doble. 21
Figura 8. Esquema del algoritmo de descompresión doble. 22
Figura 9. Ejemplo acceso a memoria. 27
Figura 10. Esquema arquitectura de descompresión simple. 29
Figura 11. Secuencia código VHDL descompresión simple. 29
Figura 12. Esquema arquitectura de descompresión doble. 31
Figura 13. Secuencia código VHDL descompresión doble 32
Figura 14. Esquema procesos descompresión doble. 34
Figura 15. Diagrama de bloques del protocolo de comunicación FSL 39
Figura 16. Diagrama de flujo de la máquina de estados de Desc_primario_1. 45
Figura 17. Diagrama de flujos del proceso Buffer1 de Desc_primario_1. 48
Figura 18. Diagrama de flujo de la máquina de estados de Desc_secundario_1. 54
Figura 19. Diagrama de flujos del proceso Buffer1 de Desc_secundario_1. 58
Figura 20. Ejemplo descompresión 2 datos. 64
Figura 21. Esquema arquitectura de compresión y descompresión doble. 66
Figura 22. Ejemplo funcionamiento diseño completo. 68
Figura 23. Ejemplo simulación Quartus II. 70
Figura 24. Diferencia entre comunicación síncrona o asíncrona. 72
Figura 25. Comparativa velocidad del descompresor entre diseños ambos en Nexys 2. 74
Figura 26. Comparativa velocidad del descompresor con distintas frecuencias de Microblaze en Nexys 2. 75
Figura 27. Comparativa velocidad del descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Nexys 2. 76
Figura 28. Comparativa velocidad del descompresor entre distintas FPGAs. 78
Figura 29. Comparativa velocidad del descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Spartan 3. 79
Figura 30. Comparativa velocidad del descompresor con distinta cantidad de datos. 80
Figura 31. Vatímetro Watts up? .Net. 82
Figura 32. Modificaciones para conectar la Nexys 2 a la alimentación a la red eléctrica. 83

Figura 33. Consumo energía del descompresor con distintos tipos de datos y diferente comunicación. 84

Figura 34. Comparación consumo energía del descompresor a distintas frecuencias de Microblaze. 84

Figura 35. Datos procesados por vatio en la FPGA Nexys 2. 85

Figura 36. Comparación consumo Nexys 2 y Spartan 3 del descompresor a distintas frecuencias de Microblaze. 86

Figura 37. Datos procesados por vatio en las FPGAs Nexys 2 y Spartan 3. 87

Figura 38. Comparativa velocidad del compresor - descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Nexys 2. 90

Figura 39. Comparativa velocidad del compresor - descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Nexys 3. 92

Figura 40. Comparativa velocidad del compresor - descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Atlys. 94

Figura 41. Comparativa velocidad del compresor - descompresor en distintas FPGAs. 94

Figura 42. Comparación consumo energía del compresor - descompresor a distintas frecuencias Microblaze con la placa Nexys 2. 95

Figura 43. Comparación consumo energía del compresor - descompresor a distintas frecuencias Microblaze con la placa Nexys 3. 96

Figura 44. Visualización interface monitorización consumo energético Atlys. 97

Figura 45. Comparación consumo energía del compresor – descompresor con distinto método de medición a distintas frecuencias Microblaze con la placa Atlys. 98

Figura 46. Comparación velocidad y consumo energético entre FPGAs. 99

Figura 47. Datos procesados por vatio en las FPGAs Nexys 2, Spartan 3 y Atlys a máxima frecuencia. 100

Figura 48. Comparativa de velocidad de descompresión hardware y software. 106

Figura 49. Comparativa de velocidad de descompresión hardware y software en FPGA. 106

Figura 50. Comparativa ciclos de reloj necesarios por dato en plataforma hardware y software. 108

Figura 51. Comparativa de datos procesados por vatio entre implementación hardware y software. 109

Figura 52. Diseño base. 115

Figura 53. Diseño objetivo. 115

Figura 54. Modificación system.mhs Microblaze 116

Figura 55. Modificación system.mhs Desc_primario_1 y creación de FSL. 116

Figura 56. Modificación Desc_primario_1_v2_1_0.mpd. 117

Figura 57. Modificación Desc_primario_1.vhd. 118

Figura 58. Modificación Desc_primario_1_v2_1_0_app.c parte 1. 118

Figura 59. Modificación Desc_primario_1_v2_1_0_app.c parte 2. 119

Figura 60. Errores primera compilación del código C. 119

Figura 61. Modificación Desc_primario_1.h 120

Figura 62. Errores segunda compilación del código C. 120

Figura 63. Modificación xparameters.h 121

LISTA DE TABLAS

Tabla 1. Ejemplo compresión simple en hexadecimal. 23

Tabla 2. Ejemplo descompresión simple en hexadecimal. 23

Tabla 3. Ejemplo compresión doble en hexadecimal. 24

Tabla 4. Ejemplo descompresión doble en hexadecimal. 25

Tabla 5. Ejemplo compresión mapeo memoria. 26

Tabla 6. Ejemplo descompresión mapeo de memoria. 26

Tabla 7. Ejemplo llenado buffer y transmisión FSL. 37

Tabla 8. Listado de señales de comunicación FSL descompresor doble. 41

Tabla 9. Comparativa velocidad del descompresor entre ambos diseños en Nexys 2. 73

Tabla 10. Comparativa velocidad del descompresor con distintas frecuencias de Microblaze en Nexys 2. 74

Tabla 11. Comparativa velocidad del descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Nexys 2. 76

Tabla 12. Comparativa velocidad del descompresor entre distintas FPGAs. 78

Tabla 13. Comparativa velocidad del descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Spartan 3. 79

Tabla 14. Comparativa velocidad del descompresor con distinta cantidad de datos. 80

Tabla 15. Consumo energía del descompresor con distintos tipos de datos y diferente comunicación. 83

Tabla 16. Comparación consumo energía del descompresor a distintas frecuencias Microblaze. 84

Tabla 17. Datos procesados por vatio en la FPGA Nexys 2. 85

Tabla 18. Comparación Nexys 2 y Spartan 3 del descompresor a distintas frecuencias de Microblaze. 86

Tabla 19. Datos procesados por vatio en las FPGAs Nexys 2 y Spartan 3. 87

Tabla 20. Comparativa velocidad del compresor - descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Nexys 2. 89

Tabla 21. Comparativa velocidad del compresor - descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Nexys 3. 91

Tabla 22. Comparativa velocidad del compresor - descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Atlys. 93

Tabla 23. Comparativa velocidad del compresor - descompresor en distintas FPGAs. 94

Tabla 24. Comparación consumo energía del compresor - descompresor a distintas frecuencias Microblaze con la placa Nexys 2. 95

Tabla 25. Comparación consumo energía del compresor - descompresor a distintas frecuencias Microblaze con la placa Nexys 3. 96

Tabla 26. Comparación consumo energía del compresor – descompresor con distinto método de medición a distintas frecuencias Microblaze con la placa Atlys. 97

Tabla 27. Comparación velocidad y consumo energético entre FPGAs. 98

Tabla 28. Datos procesados por vatio en las FPGAs Nexys 2, Spartan 3 y Atlys a máxima frecuencia. 99

Tabla 29. Comparativa de velocidad de descompresión hardware y software. 105

Tabla 30. Comparativa ciclos de reloj necesarios por dato en plataforma hardware y software. 107

Tabla 31. Comparativa de datos procesados por vatio entre implementación hardware y software. 108

Tabla 32. Costes de personal. 113

Tabla 33. Costes de hardware. 113

Tabla 34. Costes de software. 114

Tabla 35. Presupuesto total. 114

1. INTRODUCCIÓN

Este primer capítulo tiene como objetivo dar una visión global del proyecto que se ha realizado, los motivos por los que se ha dado pie a la concepción del proyecto, las motivaciones y los objetivos perseguidos.

1.1 DESCRIPCIÓN DEL PROBLEMA

Este proyecto forma parte de un trabajo futuro de un primer proyecto, es decir, es la continuación de un proyecto ya realizado, por lo que el punto de partida es el resultado obtenido en el trabajo realizado con anterioridad. Dicho proyecto consistió en implementar un algoritmo de descompresión en lenguaje de descripción hardware VHDL, en una FPGA. La idea principal del proyecto ha sido complementar esa implementación de la compresión realizando el trabajo inverso, es decir implementar un algoritmo de descompresión compatible con la compresión existente. En un primer lugar se realizó una primera implementación de una descompresión simple y más adelante se completó con una descompresión doble.

En el modelo de descompresión propuesto los datos son recibidos en un número genérico de campos distintos, que en nuestro caso será de tres, y cada campo se descomprime de forma totalmente individual. Con los datos recibidos y mediante el algoritmo de descompresión se reconstruyen los datos iniciales (antes de la compresión). Dichos datos de entrada son los datos que se obtienen a la salida del compresor.

La implementación en las distintas FPGAs que se han utilizado se ha llevado a cabo mediante el programa Xilinx Platform Studio (XPS). Este programa ha sido el entorno de desarrollo del lenguaje VHDL ya que permite la configuración de un microprocesador Microblaze programado en lenguaje C. Dicho microprocesador permite realizar pruebas de validación del diseño. También es posible añadir distintos coprocesadores que son los que se han implementado en VHDL. Todos estos módulos de microprocesadores y coprocesadores van interconectados mediante buses de comunicación FSL (Fast Simplex Link).

En último lugar el programa XPS genera los archivos necesarios que implementan la FPGA, tanto la parte hardware (VHDL) como la parte software (Microblaze), que una vez cargados en la placa permite el correcto funcionamiento.

Una vez se ha implementó el algoritmo de descompresión doble y se ha validado su funcionamiento se procedió a integrar en un mismo diseño tanto la implementación del compresor como el descompresor para verificar que tanto el algoritmo como los diseños son correctos. Por último se realizaron distintas pruebas como la velocidad de compresión – descompresión, eficiencia energética y coste hardware de los diseños.

1.2. MOTIVACIÓN

Hoy en día un aspecto muy importante es la compresión de datos ya que se manejan cantidades enormes de datos e información, por lo que su uso se hace indispensable.

La ejecución de este proyecto fin de carrera tiene **tres motivaciones principales.**

En primer lugar, dado que es un trabajo futuro de otro proyecto realizado con anterioridad que consistió en la implementación de un algoritmo de compresión, existe una necesidad de implementar de igual manera el algoritmo de descompresión correspondiente ya que, sólo contar con la compresión o la descompresión no tiene utilidad, se necesitan ambos, uno para comprimir antes de transmitir la información o almacenarla y otro para descomprimir la información una vez ha sido enviada o cuando se requiera su uso.

En segundo lugar, una vez realizada la primera etapa de implementar mediante lenguaje VHDL el algoritmo de descompresión en una FPGA y validar su funcionamiento hay que realizar la implementación conjunta de la compresión – descompresión en un único diseño para comprobar el correcto funcionamiento tanto del algoritmo como de los diseños implementados. Así una misma FGPA puede realizar la función de compresor o descompresor según convenga.

En tercer y último lugar hay que valorar si la implementación del algoritmo ha sido buena ya que este algoritmo está funcionando en la actualidad en una CPU por lo que hay que comprobar si el nuevo diseño es bueno o no, por eso surge la necesidad de realizar una evaluación lo más amplia posible, como por ejemplo velocidad de compresión – descompresión, consumo energético, pruebas en distintos dispositivos y comparación del diseño hardware con el software. Como es normal siempre se pueden hacer más pruebas pero en principio se han realizado las más importantes y significativas dado que son las que van a mostrar la información necesaria para ver qué modelos de FPGAs se adaptan mejor o si es necesario mantener la CPU o se puede reemplazar por una FPGA.

Se considera que la compresión de datos es un tema importante e interesante en muchas disciplinas, como por ejemplo comunicaciones o almacenamiento de información, por lo que se ha tratado de hacer un trabajo lo más preciso posible para intentar obtener resultados satisfactorios.

Por otra parte el uso de las FPGAs es un acierto ya que es un dispositivo que cada vez cobra más importante por el bajo coste (comparado con equipos informáticos), bajo mantenimiento, bajo consumo energético y fácil uso. Además al ser un elemento hardware y hacerse un diseño modular se puede trabajar en paralelo lo que acelera el proceso de compresión – descompresión.

1.3. OBJETIVOS

En este proyecto fin de carrera se han abordado los siguientes objetivos:

- Diseño de un algoritmo hardware en lenguaje VHDL de descompresión que se complemente con el diseño anterior de compresión.
- Implementación del algoritmo hardware en FPGA. Esta implementación debe realizarse mediante el entorno de desarrollo debe ser Xilinx Platform Studio (XPS) para realizar un proyecto que incluya el diseño para la FPGA y un microprocesador que lo controle.
- Diseño de una interfaz software en lenguaje C que permita el envío de tramas de datos conocidas y recepción de los datos descomprimidos para comprobar el correcto funcionamiento del diseño.
- Evaluación con distintos tipos de trazas con el apoyo de la herramienta Quartus II. Una vez el diseño sea operativo se procederá a comprobar su comportamiento con distintos tipos y cantidades de datos, esto permitirá ver los valores de compresión - descompresión.
- Evaluación de la arquitectura en diferentes FPGAs. Comprobar que el diseño es válido para distintos dispositivos y evaluar su funcionamiento en ellos.
- Integración del algoritmo de compresión y descompresión en un único diseño para comprobar la validez tanto del algoritmo como de los dos proyectos realizados.
- Evaluación de velocidad de compresión – descompresión en distintas FPGAs. Se ha de calcular la velocidad de procesamiento que tiene el diseño, también comprobar con que tipos de datos se consigue una mayor velocidad.
- Evaluación del consumo energético en distintas FPGAs. Se ha de comprobar el consumo que tiene el diseño en diferentes dispositivos y hacer una comparación entre ellos para ver cuál es el más eficiente.
- Evaluación de coste hardware. Se ha de verificar la cantidad de recursos de las distintas FPGAs que son precisos para el funcionamiento del algoritmo.
- Evaluación del rendimiento hardware y software. Se hará una comparación del rendimiento del algoritmo de compresión – descompresión diseñado frente al algoritmo software ya existente.
- Documentación del proceso. En último lugar se realizará un tutorial para facilitar trabajos futuros donde se aclaran las mayores dificultades a la hora de implementar el algoritmo en una FPGA. Previamente ya se ha realizado un tutorial para la rápida familiarización con el programa XPS, en este caso se hará una ampliación de éste en casos que no se habían contemplado con anterioridad.

2. ESTADO DE LA CUESTIÓN

En este capítulo se va a realizar una descripción acerca de las FPGAs, se describirán las distintas FPGAs utilizadas en este proyecto y posteriormente se hará una introducción a la compresión de datos.

2.1. INTRODUCCIÓN DE FPGAS

En el nivel más alto una **FPGA (Field Programmable Gate Array)** son chips de silicio reprogramables. Utilizan bloques de lógica preconstruidos y recursos para el rutado, gracias a esto se pueden configurar estos chips para implementar las funcionalidades deseadas en hardware. Las FPGAs son totalmente reconfigurables y pueden tener una funcionalidad totalmente distinta dependiendo del código **HDL (hardware description language)** que se haya sintetizado.

Las FPGAs ofrecen velocidades temporizadas por el hardware y fiabilidad, pero sin requerir altos volúmenes de recursos, comparado con otros lenguajes de programación. Se consigue una capacidad de ajustarse al diseño al igual que el software ya que son reprogramables. A diferencia de los procesadores, las FPGAs llevan a cabo diferentes operaciones de manera paralela, ya que cada tarea de procesos independientes se le asigna una sección dedicada del chip y puede ejecutarse de manera autónoma sin afectar a otros bloques de memoria, por lo que con menos recursos se pueden obtener rendimientos similares [NI – FPGA].

La lógica programable permite programar todo tipo de dispositivos de cómputo, desde puertas lógicas o sistemas combinacionales hasta complejos sistemas en un chip, e incluso hacerlos funcionar de forma paralela [W - FPGA] [ALT1040].

Existen varios lenguajes HDL que pueden utilizarse para realizar la descripción de un diseño. Los más utilizados y populares son Verilog, **VHDL** y SystemC. Debido a que el VHDL es de uso común en muchas instituciones universitarias y proyectos de gran importancia, es la mejor alternativa para la programación de FPGAS. Dicho lenguaje está definido por el estándar IEEE 1076 [FL – FPGA].

Las principales ventajas de las FPGAs son las siguientes. **Rendimiento**, aprovecha el paralelismo del hardware con lo que se obtiene el mismo rendimiento que un diseño software pero con menos recursos, el control de las entradas y salidas a nivel de hardware ofrece tiempos de respuesta más veloces. **Tiempo en llegar al mercado**, la tecnología FPGA ofrece flexibilidad y capacidades de rápido desarrollo de prototipos para enfrentar a los plazos de entrega, ya que cada vez hay más software para la simulación de los diseños. **Precio**, las necesidades de un sistema van cambiando con el tiempo y, como las FPGAs son reprogramables, el precio de cambiar los diseños es más bajo comparado con otros sistemas. **Fiabilidad**, mientras que las herramientas software ofrecen un entorno de programación, los circuitos de una FPGA son una implementación segura de la ejecución de un programa. **Mantenimiento a largo plazo**, los chips de las FPGAs son reprogramables en campo y no requieren el tiempo ni el precio de hacer un nuevo diseño en otro sistema, por lo que las FPGAs son capaces de mantenerse al tanto con las modificaciones del futuro [NI – FPGA].

Por otro lado, las FPGAs están limitadas por las características de la placa o de la plataforma. También es necesario un software especial diseñado por el propio fabricante, como por ejemplo, la herramienta **Xilinx Platform Studio (XPS)** diseñado por la compañía Xilinx para cargar la configuración de sus FPGAs [ALT].

Las FPGAs llevan comercializándose alrededor de 25 años, teniendo cada vez más áreas de aplicación (radioastronomía, bioinformática, criptografía, etc.). En las universidades su uso está muy extendido como una excelente herramienta didáctica [ALT1040].

Los principales fabricantes de FPGAs son **Xilinx** (www.xilinx.com) y **Altera** (www.altera.com), las cuales constan de una cuota del 80% del mercado [ALG].

Los principales bloques de los que se compone una FPGA son los siguientes:

- **CLB (bloques lógicos configurables):** se pueden programar de diversas maneras logrando así una amplia gama de funciones lógica. Cada CLB está compuesto por cuatro slices y estos a su vez contienen las llamadas LUTs (*Look up tables*), las cuales son elementos basados en memoria RAM que se pueden usar como biestables o *latches*. Las LUTs pueden tomar la forma de un bloque lógico e implementar multiplexores, o bien utilizarse como elementos de memoria donde cada una tiene una capacidad de hasta 16 bits. También puede utilizarse como un registro de desplazamiento. Las LUTs son el elemento fundamental para la síntesis de funciones lógicas.

- **IOB (bloques de entrada-salida):** son los encargados del flujo de datos desde y hasta la FPGA a través de los pines del chip. Son capaces de soportar flujos de datos bidireccionales y un total de 24 estándares de señales. También poseen control digital de impedancias.

- **BRAM o RAM de bloque:** está compuesto por varios bloques de 18 Kb. Cada cual se comporta como un chip de memoria de doble puerto. Cada puerto tiene sus propias señales de control para las operaciones de lectura y escritura.

- **Multiplicadores:** son los bloques que se dedican a efectuar estas operaciones entre dos números de 18 bits. La salida del bloque es un número de 36 bits. Se puede asociar un bloque multiplicador con un bloque de RAM, de manera que se obtiene un multiplicador síncrono con las salidas registradas. Haciendo multiplicadores en cascada es posible lograr la multiplicación de más de dos números e incluso de números de más de 18 bits.

- **DCM (controlador de reloj digital)**: estos elementos están destinados a proveer una señal de reloj de elevada exactitud. Eliminan los cambios de fase en la señal de reloj, así como las desviaciones de esta señal producto de perturbaciones externas, de altas temperaturas u otros efectos. Para esto implementan un DLL (*Delay-Locked Loop*). El DLL rastrea las desviaciones de la señal de reloj y a través de una realimentación logra eliminar el error en la señal original.

2.1.1. FPGA Nexys 2

En este proyecto se han utilizado distintas FPGAs, una de ellas es la que viene integrada en la placa Nexys 2.

La placa Nexys 2 consta de una FPGA Xilinx Spartan 3E. Esta FPGA está montada en una placa que consta con dos puertos USB, 16 MB de RAM y ROM y con varios puertos de entrada y salida. La alimentación se realiza mediante USB, siendo posible también hacerlo por transformador conectado a la red, por lo que se puede usar directamente dese un portátil [Digilent – Nexys2].

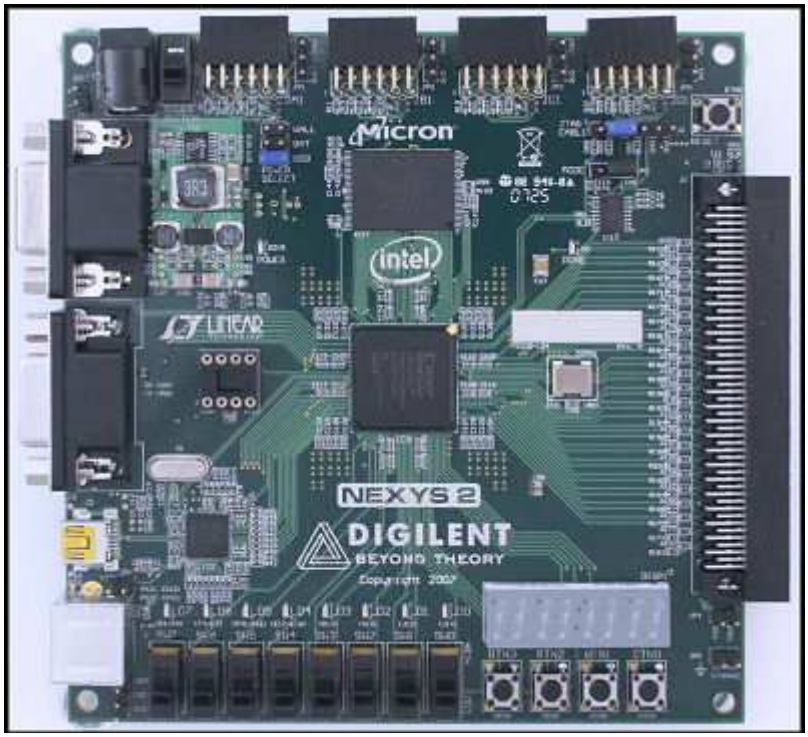


Figura 1: Placa Nexys 2.

Las principales características son las siguientes:

- 500K-gate FPGA Xilinx Spartan 3.
- USB 2.0 de alta velocidad de transmisión.
- Alimentación por USB, batería o transformador conectado a la red.
- 16 MB de RAM y 16 MB de ROM.
- Plataforma Xilinx Flash no volátil para las configuraciones de la FPGA.
- Eficiente interruptor de alimentación.
- Oscilador de 50 MHz con opción de situar un segundo oscilador.
- 60 puertos de entrada y salida para conectores de expansión.
- 8 LEDs, 4 pantallas BCD de 7 segmentos, 4 botones y 8 interruptores.
- Maletín de plástico con cable USB.

2.1.2. FPGA Spartan 3

Otra FPGA elegida ha sido la Spartan 3, integrada en el pack Spartan 3ª Starter Kit, que ofrece acceso instantáneo a un dispositivo FPGA Spartan–3A con características tales como el ahorro de energía, alta velocidad de señales de entrada y salida, interfaz de memoria SDRAM DDR2, soporte de configuración de los productos básicos de flas y la protección de FPGA / IP a través de un dispositivo de seguridad de ADN [Xilinx – Spartan3].

Figura 2: Placa Spartan-3A Starter Kit.

Las principales características son las siguientes:

- FPGA Spartan-3A.
- Plataforma Flash.
- Oscilador de 50 MHz con ranura opcional para instalar otro oscilador.
- 4 Mb Plataforma Flash PROM.
- 32M x 16 SDRAM DDR2.
- 32 Mb Flash paralela.
- 2-16 Mb dispositivos SPI Flash.
- 4 canales de convertidor D/A.
- 2 canales de convertidor A/D.
- Amplificador de señales.
- Ethernet 10/10 PHY.
- Puerto USB JTAG de descarga.
- 2 puertos RS-232 de 9 pines.
- Puerto PS2 de ratón o teclado.
- Conector VGA de 15 pines con capacidad para 4.096 colores.
- Conector FX2 de 100 pines y dos conectores de 6 pines de expansión.
- 20 señales E/S disponibles para el usuario en los pines de cabecera.
- Puerto Mini-jack estéreo de audio para PWM.
- Botón rotatorio selector de funciones.
- 8 salidas LEDs individuales.
- 4 interruptores.
- 4 pulsadores.

2.1.3. FPGA Nexys 3

En tercer lugar se ha elegido una FPGA Spartan-6 integrada en la placa Nexys 3. Esta plataforma de desarrollo dispone de la FPGA más reciente Xilinx Sparan-6, consta de 48 MB de memoria externa y los suficientes dispositivos de E/S y puertos para acoger una amplia variedad de sistemas digitales. La Nexys 3 es una plataforma ideal para que un ingeniero adquiera experiencia con las últimas tecnologías de Xilinx, ya que consta de interfaces de sencillo aprendizaje [Digilent-Nexys3].

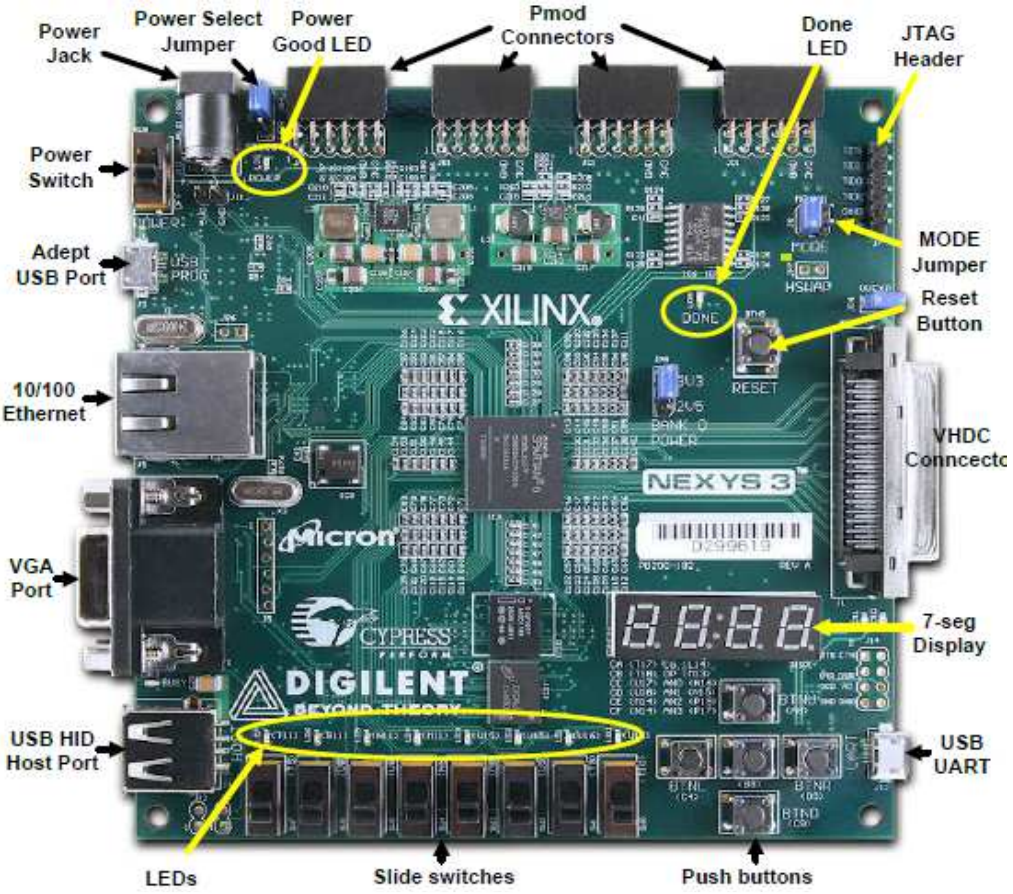


Figura 3: Placa Nexys 3.

Las principales características son las siguientes:

- FPGA Xilinx Spartan6 XC6LX16-CS324.
- 16 MB de RAM.
- 16 MB Micron paralelo PCM.
- 16 MB Micron cuádruple modo SPI PCM.
- Puerto 10/100 PHY SMSC LAN8710.
- Interfaz Digilent Adept para la transferencia de datos y programación por USB.
- Puerto USB-UART.
- Puerto USB tipo-A para ratón, teclado o tarjeta de memoria.
- Puerto VGA de 8 bits.
- Oscilador de frecuencia fija de 100 MHz.
- 8 interruptores, 4 pulsadores, 4 pantallas BCD de 7 segmentos y 8 LEDs.
- 4 conectores PMod y un conector VHDC.
- Caja de plástico resistente con cable USB.

2.1.4. FPGA Atlys

En último lugar se ha elegido la placa Atlys. La placa Atlys está lista para usar la plataforma digital de desarrollo de circuitos basado en una FPGA Xilinx Spartan 6 LX45. Consta de una colección de periféricos, incluyendo Gbit Ethernet, vídeo HDMI, 128 MB de memoria DDR2, audio, puertos USB… Es totalmente compatible con las herramientas de Xilinx por lo que los diseños pueden ser completados sin costes adicionales. Esta placa incluye la versión más reciente de Digilent Adept que a través el puerto de programación USB, se puede monitorizar a tiempo real el consumo de energía mediante una interfaz de dicho programa, pudiéndose guardar los datos obtenidos y crear una gráfica de consumo [Digilent – Atlys].

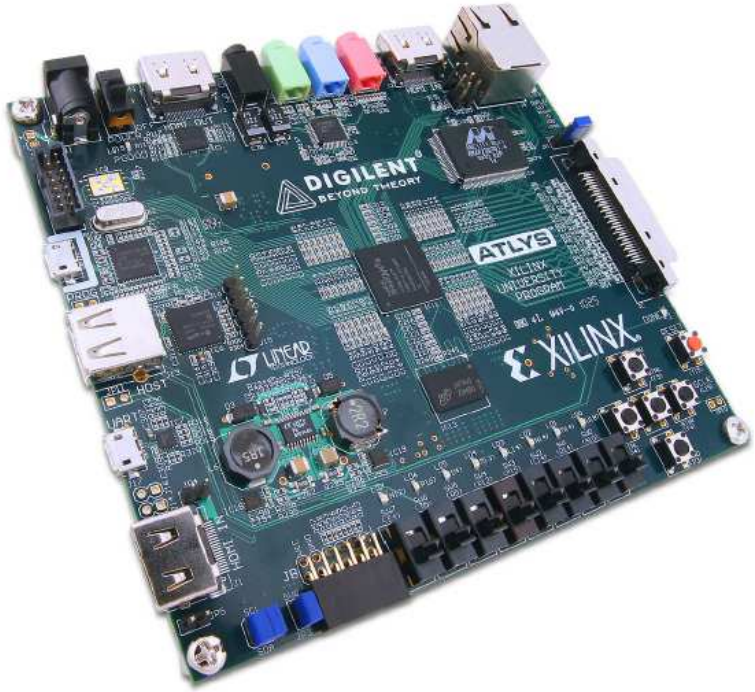


Figura 4: Placa Atlys.

Las principales características son las siguientes:

- FPGA Xilinx Spartan6 LX45, encapsulado de 324-pin BGA.
- 128 MB de memoria DDR2 de 16 bits de ancho.
- Puerto 10/100/1000 Ethernet PHY.
- 2 puertos USB para la transferencia de datos y programación USB-UART y USB-HID.
- 2 puertos HDMI de entrada de video y 2 puertos de salida HDMI.
- Codec AC-97 de entrada de línea, salida de línea, micrófono y auriculares.
- Monitorización a tiempo real del consumo energético de todas las vías de alimentación.
- 4 memorias SPI Flash de 16 MB para el almacenamiento de datos y configuración.
- Oscilador CMOS de 100 MHz.
- 48 puertos de E/S dirigida a conectores de expansión.
- 8 indicadores LEDs, 6 pulsadores, 8 interruptores.
- Caja de plástico con fuente de alimentación de 20W y cable USB.

2.2. INTRODUCCIÓN A LA COMPRESIÓN DE DATOS

Hoy en día la compresión de datos es una herramienta muy importante ya que la cantidad de datos que se manejan en cualquier ámbito, es enorme.

La compresión de datos tiene dos objetivos fundamentales, el **almacenamiento de datos** y la **reducción en el tiempo de comunicación**. Esto es posible a la reducción del volumen de datos que se almacenan o hay que transmitir. Una de las técnicas más usadas es la eliminación de datos repetitivos o que no aportan información, con lo que se puede conseguir un volumen menor de datos sin perder la información inicial.

Uno de los principales problemas de la compresión de datos es el **cómputo extra** asociado a la compresión y descompresión de datos. En el caso del almacenamiento de datos, esto no es un valor demasiado crítico, pero en el caso de la transmisión de datos, si el tiempo de compresión – descompresión, es mayor que el beneficio de tiempo obtenido al enviar menos datos, no tiene lógica utilizar este sistema, ya que no se ha logrado una reducción de tiempo y existen gastos extras dado que hay que tener un equipo de compresión y otro de descompresión.

La compresión de datos es un proceso por el que **se transforma una información a una muestra representativa**. En este proceso con la muestra representativa de la original se puede recuperar dicha información con un proceso de descompresión. En la compresión de datos, a parte de la capacidad de compresión (tamaño datos comprimidos/tamaño datos originales), hay otros parámetros que resultan de interés, como la velocidad de compresión y descompresión, espacio de memoria requerido y la generación y propagación de errores.

La compresión es un caso particular de la codificación, que tiene como característica principal que los **datos obtenidos tengan menor tamaño que los originales**. El funcionamiento se basa en buscar repeticiones en series de datos para después almacenar sólo el dato junto al número de veces que se repite. Por ejemplo un dato que conste de la siguiente cadena “1212121212” se podría comprimir como un dato 12 que se repite 4 veces, con lo que con dos valores el descompresor es capaz de descomprimir 5 datos, en caso de más repeticiones, este algoritmo tendría un rendimiento mayor.

En casos prácticos, este proceso es mucho más complejo ya que es difícil encontrar patrones tan evidentes. Por esto existen diferentes algoritmos de compresión, unos que buscan series largas que luego codifican en cadenas más cortas, por otro lado, otros algoritmos examinan los caracteres más repetidos para luego codificar de forma más corta aquellos que más se repiten, y otro caso se construyen un diccionario con los patrones encontrados a los cuales se hace referencia de manera posterior.

A la hora de hablar de compresión hay que tener presentes dos conceptos:

- **Redundancia**: Datos que son repetitivos o previsibles.
- **Entropía**: La información nueva o esencial que se define como la diferencia entre la cantidad total de datos de un mensaje y su redundancia.

La información que transmiten los datos puede ser de tres tipos:

- **Redundante**: Información repetitiva o predecible.
- **Irrelevante**: Información que no se puede apreciar y cuya eliminación por tanto no afecta al contenido del mensaje. Por ejemplo, si las frecuencias que es capaz de captar el oído humano están entre los 20 Hz y los 20.000 Hz, serían irrelevantes aquellas frecuencias que estuvieran por debajo o por encima de esos límites.

- **Básica**: La relevante. La que no es redundante ni irrelevante. La que debe ser transmitida para que se pueda reconstruir la información inicial.

Teniendo en cuenta estos tres tipos de información, se establecen tres tipologías de compresión de la información:

- **Sin pérdidas reales**: Es decir, transmitiendo toda la entropía del mensaje (toda la información básica e irrelevante, pero eliminando la redundante).
- **Subjetivamente sin pérdidas**: Es decir, además de eliminar la información redundante se elimina también la irrelevante.
- **Subjetivamente con pérdidas**: Se elimina cierta cantidad de información básica, por lo que la información se reconstruirá con errores perceptibles pero tolerables.

El objetivo de la compresión es siempre reducir el tamaño de la información, intentando que esta reducción de tamaño no afecte al contenido. No obstante, la reducción de datos puede afectar o no a la calidad de la información, por ello se pueden diferenciar dos tipos de compresión:

- **Compresión sin pérdida**: Los datos antes y después de comprimirlos son exactos en la compresión sin pérdidas. En el caso de la compresión sin pérdida una mayor compresión solo implica más tiempo de proceso. Se utiliza principalmente en la compresión de texto.
- **Compresión con pérdida**: Se puede eliminar datos para reducir aún más el tamaño, con lo que se suele reducir la calidad. Hay que tener en cuenta que una vez realizada la compresión, no se puede obtener los datos originales, aunque sí una aproximación cuya semejanza con la original dependerá del tipo de compresión. Uno de los principales campos de compresión con pérdidas es en el caso de imágenes, videos y sonidos.

3. ALGORITMO DE DESCOMPRESIÓN

En este proyecto fin de carrera se ha elegido un algoritmo de compresión con unos requisitos hardware no muy exigentes y de aplicación en escenarios reales como en el caso de la compresión en las comunicaciones. A continuación se va a proceder a explicar el algoritmo de descompresión de datos que se ha implementado en este trabajo.

3.1. DESCRIPCIÓN DEL ALGORITMO DE DESCOMPRESIÓN

La base de este proyecto es la continuación de otro proyecto anterior en el que se implementó un algoritmo de compresión, por lo tanto para realizar una implementación completa de unidades de compresión y descompresión de datos **es necesario hacer una implementación del algoritmo de descompresión**.

El proyecto inicial tenía como objetivo comprimir datos en coma flotante (32 bits divididos en tres campos). En este caso los campos que se han decidido son, un primer campo de 12 bits y dos campos de 10 bits, pero hay que destacar que **los campos son totalmente configurables número y en tamaño**. Por los campos contendrán los siguientes bits: primero de 0 a 11, segundo de 12 a 21, tercero de 22 a 31.

Se pueden distinguir claramente dos implementaciones del algoritmo de descompresión. En primer lugar se explicará una primera etapa de descompresión (descompresión simple) y a continuación las dos etapas de descompresión juntas (descompresión doble).

3.1.1. Descripción del algoritmo de descompresión simple

Para una mejor comprensión se va a realizar una breve explicación previa de en qué consiste el algoritmo de compresión que nos va a permitir posteriormente introducir el algoritmo de descompresión.

El algoritmo de compresión simple se puede resumir en el siguiente esquema:

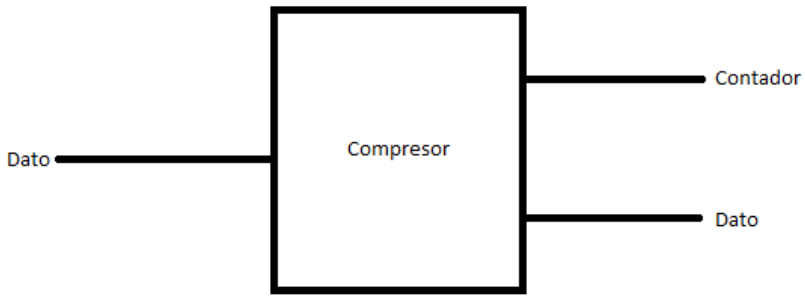


Figura 5: Esquema del algoritmo de compresión simple.

Como se puede observar en la Figura 5 la secuencia de datos se introducen en el compresor y se obtienen dos valores, la secuencia datos (que es el mismo que entra al compresor) y la secuencia de contadores, que muestra el número de veces que los datos se repiten, es decir, si dos datos consecutivos son distintos, se obtendrá dos datos de salida y el contador será cero ya que no se repite. Si dos datos consecutivos son iguales, se obtendrá un único valor de dato y el contador será uno ya que dicho dato se repite una vez. Por ejemplo en este caso los datos se dividen en dos campos de dos cifras, el primer dato es "4568" y el segundo dato es "9268" en el primer campo se obtendrán los datos "45" con contador "0" y "92" con contador "0", y en el segundo campo se obtendrá un único valor "68" y el contador "1" ya que se repiten los datos.

Por lo tanto el algoritmo de descompresión será la operación inversa de lo que se ha explicado anteriormente. El sistema contará con dos entradas y una salida. Por una parte se tendrá el valor del dato y el valor del contador y con esos dos valores habrá que reconstruir el dato original.

El algoritmo de descompresión simple se puede resumir en el siguiente esquema:

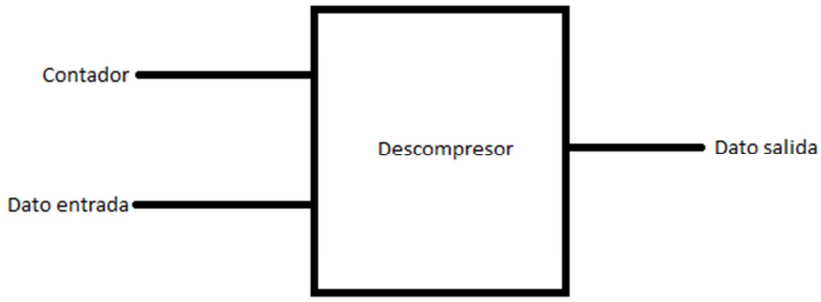


Figura 6: Esquema del algoritmo de descompresión simple.

El algoritmo se basa en contar el número de repeticiones de un dato, por lo que en vez de enviar un número "n" de veces un dato, se enviará el dato y el número "n-1" (número de veces que se repite), por lo que la transmisión será más rápida. El algoritmo de descompresión simple es fácilmente implementable. Habrá que reconstruir los datos iniciales a partir del valor del dato final y el valor del contador.

Esta explicación es válida para uno de los tres campos, por lo que, si queremos hacer una descompresión completa se necesitará tres módulos como este.

En primer lugar los datos de contador y dato de entrada son enviados por el microprocesador Microblaze mediante seis buses de **comunicación FSL a los tres coprocesadores** (uno por cada campo), la comunicación FSL se verá más adelante en el capítulo 4.3. Cada coprocesador está descrito en lenguaje VHDL. Estos datos de entrada son copiados en señales internas. Mientras que el contador sea mayor que cero copiamos en el buffer de salida el valor del dato, lo enviamos de vuelta al Microblaze y disminuimos en una unidad el valor del contador. Una vez que el contador ha llegado a cero y se han enviado todos los datos que había en el buffer de salida, los coprocesadores piden datos nuevos al Microblaze. En el apartado 3.2. se procederá a poner un ejemplo para entenderlo mejor.

3.1.2. Descripción del algoritmo de descompresión doble

De igual manera que en la descompresión simple, se va a proceder a hacer una breve introducción al algoritmo de compresión doble para que sea más fácil comprender posteriormente la descompresión.

El algoritmo de compresión doble se puede resumir en el siguiente esquema:

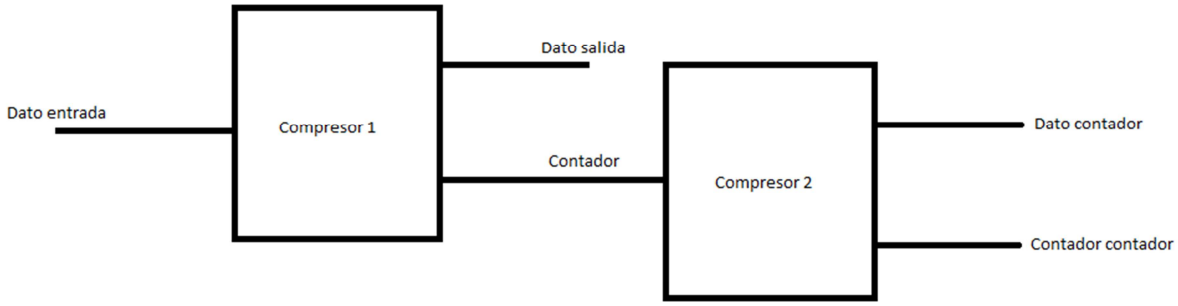


Figura 7: Esquema del algoritmo de compresión doble.

La compresión doble como su nombre indica consta de dos partes, una primera compresión que es exactamente igual a la explicada anteriormente y una segunda etapa de compresión. Esta segunda etapa hace una compresión del contador ya que si los datos de entrada son distintos, el valor del contador será cero tantas veces como datos de entrada distintos haya, por lo que el valor del contador se repite y se puede comprimir, al comprimirlo podremos transmitir menos datos con la misma información. De esta forma tendremos distinta información: el dato de entrada, el dato de la primera compresión, el contador de la primera compresión (que se volverá a comprimir), el dato de la segunda compresión (que llamaremos dato del contador) y por último el contador de la segunda compresión (que llamaremos contador del contador). Por ejemplo si los datos de entrada son “12”, “58” y “69” (en este caso con un único campo), los datos de salida serán “12”, “58” y “69” y el contador “0”, “0” y “0”, que después de la segunda compresión quedará, dato de contador “0” y contador de contador “2”.

Como se puede observar en la Figura 7 el algoritmo de compresión doble consta de **dos compresiones simples** idénticas. Como ya se ha dicho antes obtendremos tres valores, del primer compresor el valor real del dato y del segundo compresor los valores del dato y contador correspondientes al contador de la primera compresión, los cuales serán necesarios más adelante para su posterior descompresión.

Este método permite transmitir datos a mayor velocidad ya que para un mismo número de datos, con la compresión doble se necesitan enviar menos valores que en la compresión simple.

Al igual que en el caso anterior esta explicación es válida para un único campo, por lo tanto, al haber diseñado tres campos habrá que utilizar tres pares de módulos como los que se acaban de describir.

Una vez comprendida la parte de la compresión es más fácil entender la descompresión. Lo único que hay que hacer es invertir el algoritmo para poder reconstruir los datos correctamente.

El algoritmo de compresión doble se puede resumir en el siguiente esquema:

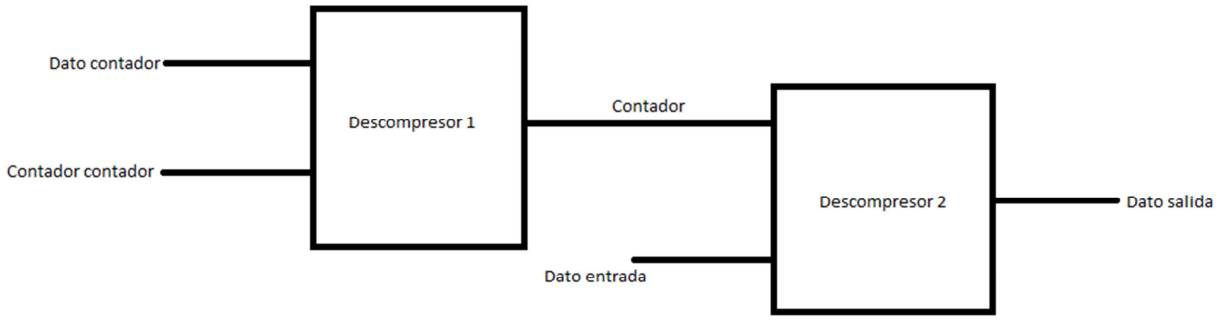


Figura 8: Esquema del algoritmo de descompresión doble.

El esquema es muy parecido al de la descompresión simple, la única diferencia es que antes de realizar la descompresión del dato hay que realizar la descompresión del contador, que de eso se encarga la primera etapa de descompresión. Dicha etapa funciona de igual manera que la descrita anteriormente, es decir el microprocesador Microblaze envía a tres coprocesadores descritos en lenguaje VHDL los valores que se han llamado "**Dato Contador**" y "**Contador de contador**". El valor de "Contador de contador" se copia en una señal interna, siempre que el valor de dicha señal sea mayor que cero, el coprocesador enviará al buffer de salida el valor del "Contador" y lo enviará a la segunda etapa de descompresión. En caso de que el valor de la señal llegue a cero se enviará el último "Contador" y mandará una señal para recibir nuevos datos de entrada". Una vez que se envían los valores de "Contador" a la segunda etapa ésta es la descrita en el apartado de descompresión simple. Por ejemplo, si el dato de contador es "25" y el contador de contador es "3", quiere decir que el contador de valor "25" se repite tres veces, por lo que después de la descompresión el valor de contador será "25", "25", "25" y "25" que pasará a la segunda etapa de descompresión que reconstruirá la información inicial con dicho contador y los datos de entrada transmitidos.

En el caso de la descompresión **la primera etapa sirve para reconstruir el valor del contador** para poder seguir con el proceso, este dato es después utilizado en **la segunda etapa de la descompresión que junto con el valor del dato permite reconstruir el dato original**. Como ya se ha dicho anteriormente esta descompresión doble permite una mayor rapidez en la transmisión de información ya que hay que enviar una menor cantidad de datos.

3.2. EJEMPLOS DE FUNCIONAMIENTO

A continuación se va a proceder a explicar unos ejemplos para facilitar y asegurar el entendimiento de los algoritmos de compresión y descompresión respectivamente.

3.2.1. Ejemplo funcionamiento descompresión simple

En primer lugar se explicará un ejemplo de compresión simple:

Dato entrada	Dato campo 1	Dato campo 2	Dato salida campo 1	Contador salida campo 1	Dato salida campo 1	Contador salida campo 1
2576	25	76	25	2	76	0
2569	25	69			69	0
2513	25	13			13	1
8113	91	13	91	0		
5427	54	27	54	2	27	2
5427	54	27				
5427	54	27				

Tabla 1: Ejemplo compresión simple en hexadecimal.

Como se puede observar en el ejemplo en primer lugar se introducen siete datos, estos datos son divididos en los correspondientes campos. En el caso del primer campo, los tres primeros datos “25” se repiten 2 veces, por lo que el dato de salida será del mismo valor y el contador valdrá “2”, el siguiente dato “91” no se repite y en el último caso el dato “54” se repite 2 veces. En el segundo campo, el primer dato “76” no se repite, el segundo dato “69”, el tercer dato “13” se repite una vez, y por último el cuarto dato se repite 2 veces (recordemos que el valor de contador es el valor del número de veces que se repite el dato), como se introducen tres datos iguales, el primer dato se repite dos veces, por lo que el contador toma ese valor.

Con este ejemplo se muestra que en caso que los datos se repitan se obtiene una gran mejora ya que solo es necesario enviar dos datos, y en caso que no se repitan los datos de entrada no se consigue compresión alguna.

Por lo tanto la descompresión simple será de la siguiente manera:

Dato entrada campo 1	Dato entrada campo 1	Contador campo 1	Contador campo 2	Dato salida
25	2	76	0	2576
		69	0	2569
		13	1	2513
81	0			8113
54	2	27	2	5427
				5427
				5427

Tabla 2: Ejemplo descompresión simple en hexadecimal.

En este caso son los mismos datos que en el ejemplo de la compresión simple. Empezando por el primer campo, en primer caso se tiene un dato de entrada ”25” que se repite 2 veces, por lo que el dato de salida (el original antes de la compresión) tiene el mismo valor en tres ocasiones consecutivas.

El segundo caso, es un dato “81” que no se repite ninguna vez por lo que el dato de entrada es el mismo que el de salida.

Por último tenemos un dato “54” que se repite dos veces y al igual que en el primer caso, ese es el valor del dato de salida en tres ocasiones simultáneas.

3.2.2. Ejemplo funcionamiento descompresión doble

De igual manera que en la descompresión simple primero se planteará un ejemplo de la compresión doble para tener una vista más sencilla para la posterior descompresión. En este caso el ejemplo es el siguiente:

<table>
<tr><th>Dato entrada</th><th>Dato salida</th><th>Contador</th><th>Dato de contador</th><th>Contador de contador</th></tr>
<tr><td>25</td><td rowspan="4">25</td><td rowspan="4">3</td><td rowspan="4">3</td><td rowspan="4">0</td></tr>
<tr><td>25</td></tr>
<tr><td>25</td></tr>
<tr><td>25</td></tr>
<tr><td>12</td><td>12</td><td>0</td><td rowspan="6">0</td><td rowspan="6">5</td></tr>
<tr><td>87</td><td>87</td><td>0</td></tr>
<tr><td>63</td><td>63</td><td>0</td></tr>
<tr><td>31</td><td>31</td><td>0</td></tr>
<tr><td>58</td><td>58</td><td>0</td></tr>
<tr><td>10</td><td>10</td><td>0</td></tr>
<tr><td>84</td><td rowspan="2">80</td><td rowspan="2">1</td><td rowspan="2">1</td><td rowspan="2">1</td></tr>
<tr><td>84</td></tr>
</table>

Tabla 3: Ejemplo compresión doble en hexadecimal.

La primera parte del ejemplo, es decir, la obtención de “**Dato salida**” y “**Contador**” es de la misma manera que en la compresión simple y lo que se añade en la compresión doble es que se hace una segunda compresión pero esta vez del contador obtenido tras la primera compresión, para que sea más sencillo entenderlo y para que la tabla tenga dimensiones aceptables se va a realizar el ejemplo con un único campo ya que en la compresión y descompresión simple ya se ha visto el funcionamiento con varios campos.

En este caso como se puede observar en primer lugar los primeros cuatro datos “25” de entrada se comprimen y en la segunda compresión no se consigue nada ya que han sido comprimidos con anterioridad.

En cambio en los posteriores casos (del segundo al séptimo, los datos “12”, “87”, “63”, “31”, “58”, y “10”) son datos que no se repiten por lo que la primera compresión no aporta nada, en cambio, al no repetirse, el valor de los contadores son todos “0” por lo que en este caso se podrá hacer una segunda compresión del contador “0” permitiendo enviar un menor número de contadores sin perder información.

Por último el octavo “84” caso es idéntico al primero ya que son datos que se repiten consecutivamente por lo que el contador no puede ser comprimido.

Como puede observarse en el ejemplo, si no se comprimieran los datos habría que transmitir **doce datos**, mientras que con la compresión doble con transmitir **ocho datos, tres datos de contador y tres contadores de contadores** son suficientes y no se pierde información.

La **segunda compresión es realmente útil si los valores de entrada son distintos** ya que no hay repeticiones y el valor del contador será cero, por lo que al hacer la segunda compresión será necesario enviar muchos menos datos para la correcta reconstrucción de los mismos. En ese caso el primer compresor no será demasiado útil ya que si no se repiten los datos no habrá compresión.

Por otro lado la **primera compresión resulta ciertamente eficiente cuando los datos de entrada se repiten** un gran número de veces, en ese caso tendremos menos datos de salida para transmitir, y al contrario que antes, la segunda compresión no será tan productiva.

En conclusión el sistema tiene una primera etapa de compresión que resultará útil en el caso que se repitan los datos y una segunda etapa de compresión que será beneficioso en caso que no se repitan los datos por lo que ambas etapas son muy interesantes ya que se complementan perfectamente.

Para terminar se verá un ejemplo ilustrativo en el que se podrá ver con mayor detalle las ventajas de la doble descompresión y se podrá comprender con mayor facilidad:

Dato de contador	Contador de contador	Contador	Dato entrada	Dato salida
0	4	0	25	25
		0	12	12
		0	87	87
		0	63	63
3	0	3	31	31
				31
				31
				31
0	1	0	58	58
		0	10	10
2	0	2	84	84
				84
				84

Tabla 4: Ejemplo descompresión doble en hexadecimal.

Este ejemplo es similar al de la compresión doble, en este caso se parte de dos valores “Dato contador” y “Contador de contador” con los que se reconstruye el valor “Contador” con el que, junto con el valor “Dato” se obtendrán los valores originales de los datos.

En el primer caso tenemos un valor de contador “0” que se repite tres veces, es decir, hay cuatro datos consecutivos que son distintos, o lo que es igual, hay cuatro contadores de valor cero.

El segundo caso es la otra posibilidad que existe. Esta vez se parte de un valor de contador "3" que no se repite, eso quiere decir que el contador para la segunda descompresión tiene valor de "3" y es único. Con ese valor en la segunda descompresión se obtienen cuatro datos iguales consecutivamente.

El resto de casos son iguales a estos dos que se acaban de explicar, el tercer caso es igual que el primero y el cuarto que el segundo.

Como se observa en el ejemplo con sólo **cuatro valores de dato de contador y contador de contador** podemos reconstruir **trece valores de dato** de salida. En el ejemplo se han puesto valores variados, es decir valores con ciertas repeticiones y valores sin repeticiones para mostrar que con este tipo de compresión-descompresión se obtiene un notable ahorro en el número de datos a enviar, en este caso con mandar 8 datos se pueden descomprimir 13 datos originales, lo que lógicamente repercute positivamente en la velocidad de transmisión de los datos ya que enviamos menos datos pero sin pérdida de información.

Más adelante comprobaremos que se obtiene una gran mejora en cuanto a velocidad de transmisión de datos.

Básicamente estos son los dos algoritmos de descompresión que se han utilizado en este proyecto (la simple y la doble). Esta es la base del proyecto y es el punto de partida del mismo. En primer lugar se realizó un primer diseño con el algoritmo de descompresión simple, una vez hecho esto se comprobó el correcto funcionamiento del diseño. Posteriormente se amplió el diseño a la descompresión doble hasta conseguir el funcionamiento deseado.

A continuación se va a realizar una explicación para demostrar como resultarían estos ejemplos en el mapeo de memoria de Microblaze, el ejemplo se va a realizar para un proceso de compresión – descompresión simple pero es extensible al proceso doble.

Datos entrada		Datos salida		Contadores salida	
Campo 1	Campo 2	Campo 1	Campo 2	Campo 1	Campo 2
A	F	A	D	3	0
A	1		1		0
A	2		2		1
A	2				

Tabla 5: Ejemplo compresión mapeo memoria.

Este es un caso general en el que se realiza una compresión de datos, en este caso el campo uno tiene un único valor "A" que se repite 3 veces, y el segundo campo tiene varios valores diferentes "F", "1" y "2". Por lo tanto el primer campo sólo tendrá un valor de dato de salida y otro de contador de salida mientras el campo dos constara de 3 de cada uno. A la hora de la descompresión pasará lo siguiente:

Datos salida	
Campo 1	Campo 2
A	F
A	
A	
A	

Datos salida	
Campo 1	Campo 2
A	F
A	1
A	
A	

Datos entrada	
Campo 1	Campo 2
A	F
A	1
A	2
A	2

Tabla 6: Ejemplo descompresión mapeo de memoria.

Como se observa en la memoria de Microblaze se van almacenando los datos correspondientes a la descompresión, como el campo 1 sólo tiene un dato ”A” descomprime 4 datos y los copia en memoria mientras que el campo 2 tiene que hacer esto tres veces “F”, “1” y “2” para descomprimir los 4 datos, por lo que va rellenando las posiciones de memoria según son descomprimidos. El número total de datos descomprimidos tiene que ser el mismo en todos los campos ya que los dos campos del ejemplo (en el caso real tres) son una parte de un dato que engloba todos los campos. Estos procesos de mapeo de memoria se ejecutan en paralelo por cada campo y son totalmente independientes.

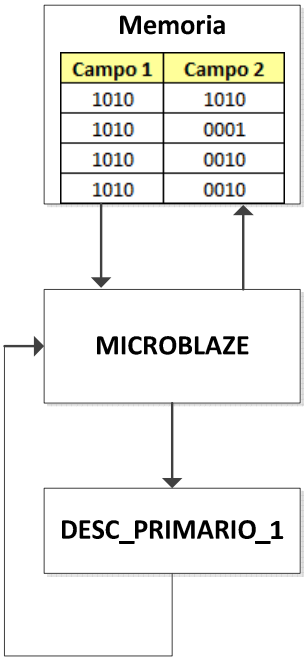


Figura 9: Ejemplo acceso a memoria.

Como se puede observar en la Figura 9, se corresponde al ejemplo anterior pero en vez de en hexadecimal, esta vez en binario. En este caso se comprueba que el acceso de la memoria es totalmente independiente en cada campo, es decir va rellenando como en el ejemplo anterior hasta que termina, en ese caso todos los campos tienen que tener el mismo número de datos, aunque durante el proceso un campo puede tener más datos que otro, por lo que las zonas de memoria distintas pero contiguas. En el caso real se han implementado tres campos distintos genéricos que se podrían ajustar a la norma IEEE 754.

El estándar para coma flotante (IEEE 754) es el más extendido, y es seguido por muchas de las mejoras de CPU y FPU. El estándar define formatos para la representación de números en coma flotante y valores desnormalizados, así como valores especiales. También especifica cuatro modos de redondeo y cinco excepciones.

IEEE 754 especifica cuatro formatos para la representación de valores en coma flotante: precisión simple (32 bits), precisión doble (64 bits), precisión simple extendida (≥ 43 bits) y precisión doble extendida (≥ 79 bits). Sólo los valores de 32 bits son requeridos por el estándar, los otros son opcionales. Muchos lenguajes especifican qué formatos y aritmética de la IEEE implementan, a pesar de que a veces son opcionales.

4. DISEÑO HARDWARE

En este capítulo se va a proceder a explicar con distintos niveles de detalles el diseño hardware del algoritmo de descompresión explicado en el capítulo anterior.

El diseño se ha realizado mediante un lenguaje de descripción hardware (VHDL) que básicamente describe un circuito hardware a partir de un lenguaje parecido a algunos lenguajes de programación.

En primer lugar se va a proceder a explicar las arquitecturas que se han diseñado, a continuación se hará una explicación general del funcionamiento de cada módulo. En tercer lugar se introducirá más en detalle en la programación de los módulos. Por otra parte se explicará la interfaz software que se ha usado para probar el correcto funcionamiento del diseño y para hacer las pruebas de rendimiento. En último lugar se ha **integrado** el proyecto inicial (compresor) con la continuación (descompresor) en una misma FPGA para validar ambos proyectos y tener un diseño versátil ya que puede comprimir o descomprimir según la necesidad existente.

4.1. ARQUITECTURAS DISEÑADAS

El objetivo de este proyecto de fin de carrera era complementar otro proyecto realizado con anterioridad que trataba de una compresión doble, por lo que el objetivo principal de este proyecto es alcanzar un **pleno funcionamiento** de un descompresor doble.

Para alcanzar el objetivo con la mayor efectividad posible se realizó un **proceso progresivo** que constó de empezar con un diseño sencillo y evolucionarlo poco a poco hasta conseguir la doble descompresión, por ello se pueden distinguir dos arquitecturas, la descompresión simple y la descompresión doble. La descompresión doble no es más que dos descompresores simples similares que se complementan.

El número de campos es genérico y sólo queda limitado por el número de FSLs que el entorno de desarrollo XPS permite conectar al Microblaze. En nuestro diseño como caso de estudio se han escogido 3 campos para todo el estudio.

A continuación se va a proceder a explicarlas para tener una visión general de su arquitectura.

4.1.1. Arquitectura de descompresión simple

La arquitectura de descompresión simple fue la primera arquitectura en diseñarse ya que era la más sencilla de implementar. De esta manera se podía llegar a una solución rápida y probar el correcto funcionamiento del algoritmo y la implementación, permitiendo detectar errores de diseño rápidamente y poderlos solucionar para no arrastrarlos a arquitecturas más complejas en las que es más difícil encontrar el motivo de errores de funcionamiento.

También se decidió hacer un diseño modular por las mismas razones ya que se pueden independizar los módulos entre sí para obtener un diseño más flexible ya que si en el futuro se quiere realizar algún cambio o modificación resulte sencilla y rápida.

Una vez descrita la forma en la que se ha trabajado, se va a mostrar un esquema general de la arquitectura de descompresión simple:

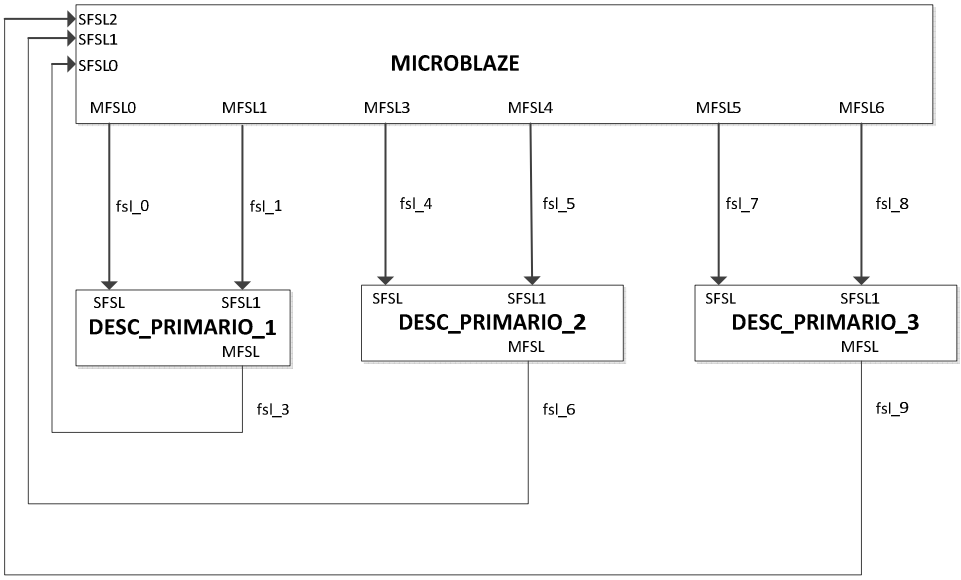


Figura 10: Esquema arquitectura de descompresión simple.

Como se observa en la Figura 10 es un diseño que consta de cuatro módulos, el Microblaze y **tres coprocesadores independientes que trabajan en paralelo**, cada uno de ellos realiza una compresión simple de un campo del dato. El Microblaze está conectado a cada coprocesador mediante dos FSL por los que se enviarán los datos que tienen que descomprimir y cada coprocesador devuelve el dato descomprimido de su campo por un único FSL al Microblaze. Los tres módulos son similares, únicamente cambian el tamaño de los campos que procesan.

Un esquema representativo del modo de funcionamiento sería el siguiente:

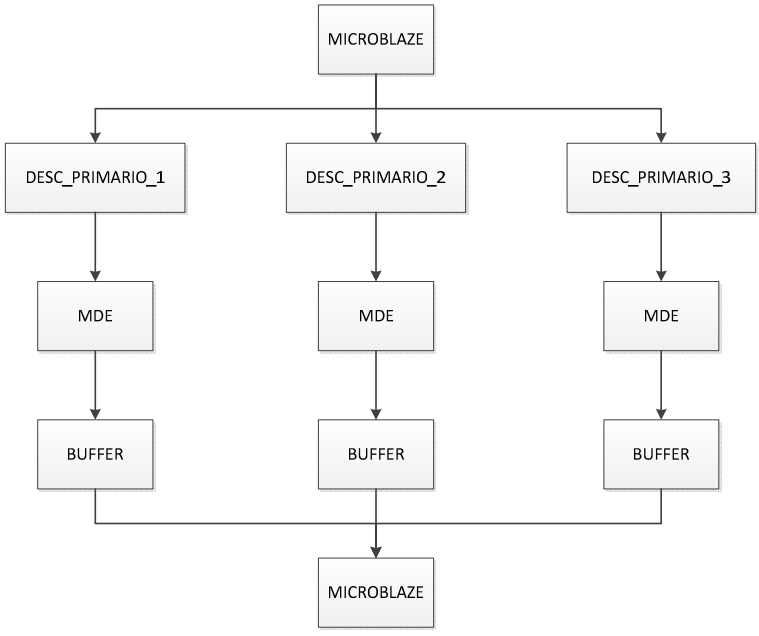


Figura 11: Secuencia código VHDL descompresión simple.

El Microblaze manda a cada módulo los valores de “Dato” y “Contador” de 32 bits cada uno al coprocesador correspondiente. Cada coprocesador tiene una máquina de estados que lee los datos de entrada, compara si el valor del contador es mayor que cero, en caso positivo manda al buffer el valor de “Dato”, y disminuye en una unidad el valor de “Contador”, este proceso se realiza hasta que contador es igual a cero entonces se envía por última vez el valor “Dato” y vuelve a estar a disposición de leer nuevos datos.

Cada coprocesador tiene dos procesos, uno es “MDE” (máquina de estados) principal en el que se recibe, y se comparan los valores y un proceso “Buffer” en el que se gestiona el envío de datos según los resultados obtenidos en “MDE”.

4.1.2. Arquitectura de descompresión doble

La segunda arquitectura incluye una segunda etapa de descompresión en la que primero habrá que descomprimir el contador y con ese valor se podrá hacer la descompresión del dato.

El esquema general de la arquitectura es el siguiente:

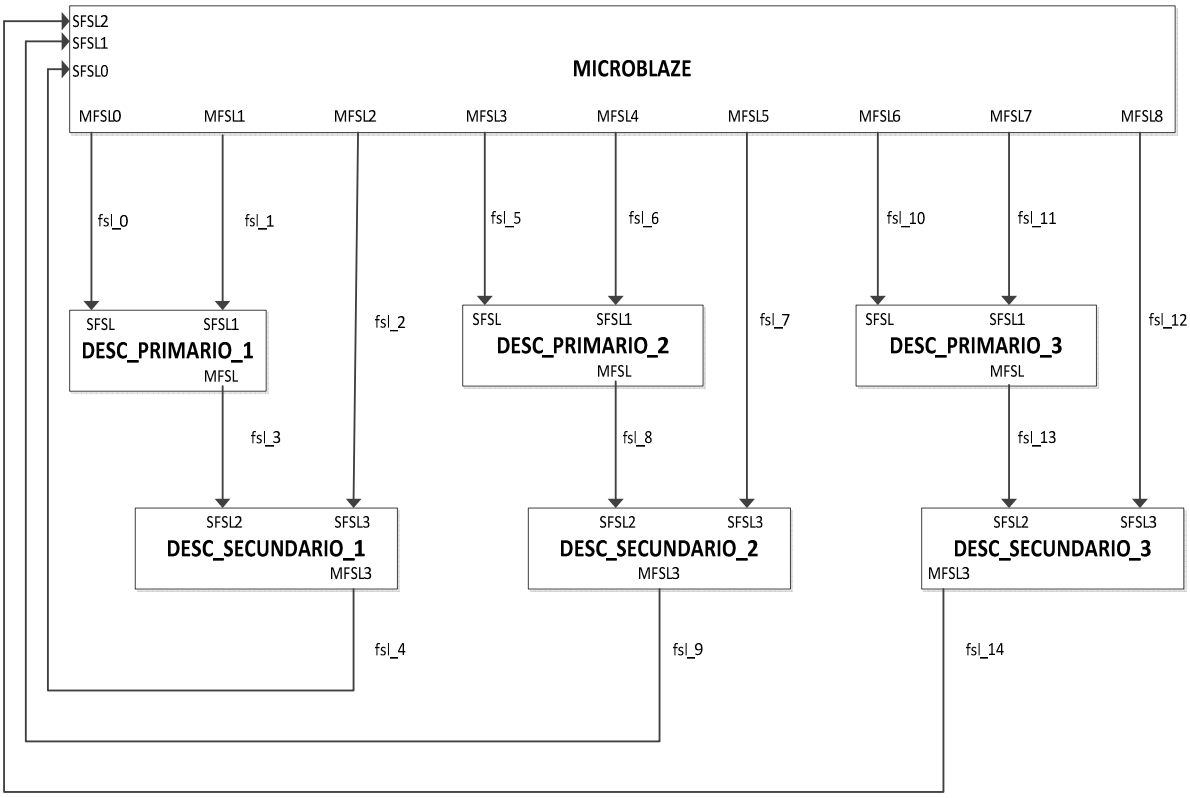


Figura 12: Esquema arquitectura de descompresión doble.

El esquema de la descompresión doble consta de siete módulos, el primero se trata del Microblaze, los tres siguientes son los debidos a la primera descompresión y los tres últimos los correspondientes a la segunda descompresión.

Como en el caso anterior cada módulo de la primera descompresión está conectado mediante dos FSLs al Microblaze. La diferencia es que en este caso la salida de los módulos de la primera descompresión no vuelve a Microblaze sino que se envían a la segunda descompresión, que se trata del valor de "Contador" necesario en estos módulos. En este punto necesitan también que el Microblaze mande el valor de "Dato" para reconstruir el dato inicial (antes de la compresión). Por lo tanto esta arquitectura consta de 15 FSLs de comunicación.

En este diseño la segunda etapa de descompresión es la correspondiente a la descompresión simple, lo único que se ha hecho ha sido añadir otra etapa de descompresión anterior que con los valores de "Contador de contador" y "Dato Contador" como entradas, obtiene el valor "Contador" necesario para la segunda etapa que junto al valor "Dato" descomprime el valor inicial.

En este diseño **se ha añadido un sistema** para que la segunda etapa sepa cuando se está procesando el último dato ya que si se ha procesado el último dato y el buffer de salida no se ha llenado éste no se enviará. Ese sistema es el siguiente, antes de enviar ningún dato, Microblaze envía el número de datos que se va a procesar, la primera etapa procesa los datos hasta que se llega al número de datos que Microblaze ha enviado con anterioridad, en ese momento se activa una señal del bus de comunicación FSL que comunica la primera y la segunda etapa. Esta señal indica que se el dato que se va a procesar es el último y si el buffer de salida no se llena se rellenará con ceros por la derecha y se enviará.

Un esquema representativo del modo de funcionamiento sería el siguiente:

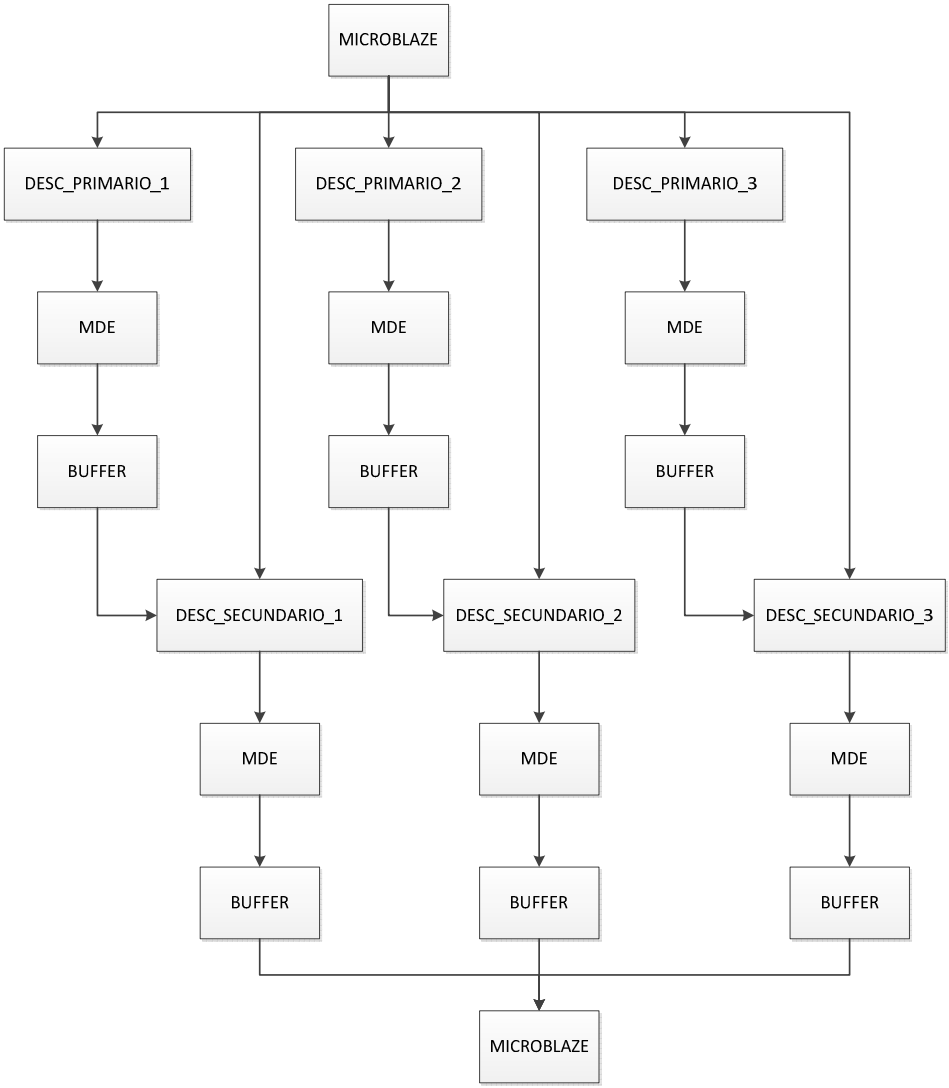


Figura 13: Secuencia código VHDL descompresión doble

Los cuatro coprocesadores son muy similares con la única diferencia que en la primera descompresión se descomprime el contador y en la segunda descompresión con ese contador y un dato enviado por Microblaze se realiza la segunda descompresión. Como ya se ha dicho los primeros tres módulos tienen que avisar a los tres siguientes cuándo se está procesando el último dato. Los valores de los contadores son como en la descompresión simple de 32 bits, como en la comunicación entre la primera y segunda descompresión se transmiten contadores también son de 32 bits, por lo que aunque los datos sean de 12 o 10 bits (según el campo) Microblaze solo envía un dato por cada conjunto de 32 bits, es decir envía los bits que contienen la información y el resto se rellena con ceros. Esto se hizo porque al ser un proceso continuo si los contadores y los datos tienen el mismo tamaño siempre se necesitarán el mismo número de ambos mientras que si en una transmisión se envían más datos que contadores el sistema no está equilibrado.

En el caso de la segunda descompresión, el mensaje de vuelta a Microblaze es un único dato por lo que se va llenando el buffer de 32 bits con datos de 12 o 10 bits hasta que se llena y se envía, por eso en este caso sí que es necesario saber cuándo se está procesando el último dato ya que el buffer de salida puede no haberse llenado. Con lo cual el proceso “Buffer” dentro de los coprocesadores de la segunda descompresión es algo más complejo, por lo demás los seis coprocesadores son similares, por lo que cuentan con un proceso “MDE” y otro “Buffer”.

4.2. DESCRIPCIÓN DE LAS ARQUITECTURAS EN ALTO NIVEL

En este apartado se pretende dar una idea global del funcionamiento completo de los diseños implementados en cada coprocesador.

4.2.1. Visión global del algoritmo

A continuación se va a mostrar un esquema de los procesos que siguen las máquinas de estado y procesos de los coprocesadores:

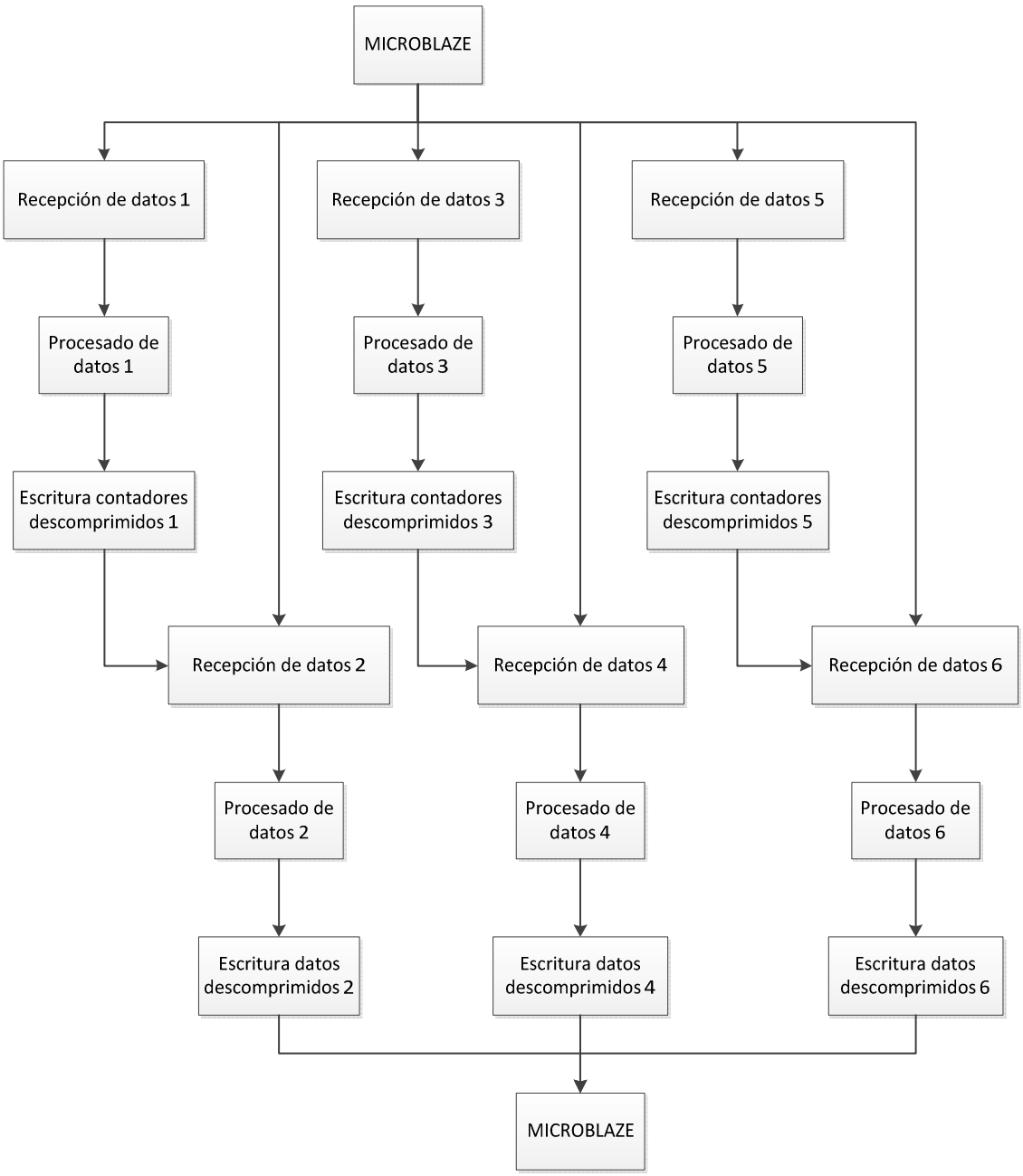


Figura 14: Esquema procesos descompresión doble.

El algoritmo de descompresión comienza con una máquina de estados principal que se encarga de hacer la primera etapa de descompresión. En primer lugar se encarga de hacer la **recepción de contadores** desde Microblaze, una vez éstos son recibidos se encarga de **procesar los contadores** (en el caso de la primera descompresión reconstruye los valores de los contadores) y en último lugar se encarga de **mandar al buffer los contadores descomprimidos**. Por otra parte existe un proceso que funciona independiente de la máquina de estados que según en el estado en el que se encuentre, llena el buffer de salida con los datos recibidos y cuando el buffer se llena los datos se envían a la segunda etapa de descompresión. También es el encargado de activar una señal en caso de que se mande el último dato.

Hay que tener en cuenta que esto que se acaba de explicar es lo correspondiente a una única rama de la primera etapa de descompresión. El diseño cuenta con tres ramas similares con tamaños de campos distintos, 12 bits, 10 bits y 10 bits que son independientes y trabajan en paralelo para conseguir un rendimiento mayor.

Una vez realizada la primera descompresión se pasa a los tres siguientes coprocesadores que se encargan de la segunda descompresión. En este caso se vuelve a tener una máquina de estados principal. En primer lugar se encarga de la **recepción de los contadores** del coprocesador de la primera descompresión, a continuación **recibe el valor de los datos** desde Microblaze, al igual que en la primera etapa, la máquina de estados se encarga de **procesar los datos recibidos**, una vez éstos son procesados, se encarga de **enviar al buffer los datos descomprimidos**. En caso de tratarse del último dato, la máquina de estados recibirá la señal de oportuna desde la primera etapa y se encargará de procesar ese último dato correctamente. De igual manera que en la descompresión simple existe un proceso independiente que se encarga de llenar el buffer y cuando éste está lleno lo envía de vuelta al Microblaze. En caso de tratarse del último dato recibirá una orden desde la máquina de estados principal de mandar los datos aunque no se haya llenado el buffer.

De igual manera se ha explicado una única rama. La arquitectura completa consta de tres descompresiones dobles como la que se acaba de describir.

4.2.2. Descompresión simple

Como se ha visto con anterioridad la descompresión simple consta de un Microblaze y tres coprocesadores. Cada coprocesador recibe un valor de contador y de dato, lo procesa y lo manda de vuelta al Microblaze. En cada coprocesador se pueden diferenciar dos procesos distintos, un primero que se trata de una máquina de estados encargada de procesar los datos llamada "MDE" y otro que se encarga de almacenar y mandar los datos de vuelta al Microblaze llamado "Buffer".

En primer lugar, la "MDE" recibe desde Microblaze el número de datos que se van a procesar. En segundo lugar recibe el valor del contador y en tercer lugar recibe el valor del dato. Esta es la información necesaria para empezar a descomprimir el primer dato, una vez que ya haya sido descomprimido no habrá que leer el número de datos que se van a procesar, sino que recibirá directamente el valor del contador y el valor de dato. Es decir, **el valor de número de datos que se van a procesar solo se recibe la primera vez que se inicia la "MDE".**

Cuando la "MDE" recibe un valor lo copia en una señal interna del sistema, es decir, los valores de número de datos que se van a procesar, contador y dato, son copiados para no perder esa información. A continuación se compara el valor de contador, si este es mayor de cero es que significa que el dato se repite, por lo que se envía el valor del dato al proceso "Buffer" y se resta una unidad al valor del contador, este proceso se repite siempre y cuando el contador sea mayor que cero. Una vez que el contador es cero, quiere decir que ya no hay más repeticiones, por lo que ya no se manda más veces el valor de contador al proceso "Buffer" y se vuelve a hacer una recepción de contador y de dato. Cada vez que se realiza una recepción de datos se aumenta el valor de un contador interno para saber cuántos datos se han procesado. Todo esto, recepción de datos y procesado de datos, se realizará hasta que el valor del contador interno sea menor que el valor de número de datos que se van a procesar. Cuando estos dos valores son iguales, significa que ya se han procesado todos los datos y que ya no se van a recibir más por lo que se activa una señal interna que el proceso "Buffer" interpretará posteriormente.

El proceso "Buffer" es el encargado de recibir los datos desde la "MDE" y almacenarlos en el buffer de salida y cuando la "MDE" lo indique mandar los datos. Cada vez que la "MDE" procesa un dato y lo envía al proceso "Buffer", éste lo almacena en el buffer de salida e incrementa una señal interna en 12 o 10 (según si estemos en el coprocesador que tiene tamaño de campo 12 bits o 10 bits), realiza esta operación hasta que el valor de esa señal es mayor de 32 (el bus de comunicación es de 32 bits), en ese momento envía los 32 primeros bits y una vez son enviados hace un desplazamiento para que el primer bit que no se ha enviado pase a ser el primer bit del próximo envío. En ese momento el valor de la señal interna que indica donde se ha de copiar los datos es disminuido en 32 unidades para que el próximo dato que almacene en el buffer coincida con el primer hueco del buffer, ya que si se resetea el valor de dicha señal empezaría a copiar los datos desde el principio y podría escribir encima de un dato útil. Poniendo un ejemplo práctico si el buffer se llena con tres datos de 12 bits hace un total de 36 bits, como el ancho del FSL es de 32 bits, se mandan los primeros 32 bits y los 4 restantes son los primeros en la siguiente transmisión, por lo que el cuarto dato a copiar en el buffer se deberá hacer después de los 4 bits que no se enviaron en el mensaje anterior.

A continuación se va a exponer un ejemplo para una mejor comprensión:

Entrada buffer	Buffer	Salida FSL
12 bits	44 bits	32 bits
485	48500000000	
6A1	4856A100000	
308	4856A130800	4856A130
E09	8E090000000	
5C6	8E095E60000	
746	8E095E67460	8E095E67
2F8	462F8000000	
D55	462F8D55000	462F8D55

Tabla 7: Ejemplo llenado buffer y transmisión FSL.

Como se puede observar en la Tabla 7 el buffer de 44 bits se va llenando con los valores de entrada, hay que aclarar que en el ejemplo se utilizan valores hexadecimales por lo que los datos de tres cifras corresponden a 12 bits. Una vez que el buffer supera los 32 bits, los 32 primeros son enviados por el FSL de salida y los restantes son desplazados para ocupar el principio del siguiente envío.

En el caso de tratarse del último dato, el proceso “Buffer” recibe una señal de la “MDE” que le indica que no se van a procesar más datos por lo tanto habrá que enviar el buffer aunque no se haya llenado porque si no se quedara esperando datos nuevos hasta que éste se llene. En este caso los bits que no se hayan quedado vacíos se rellenarán con ceros y se enviará el mensaje.

4.2.3. Descompresión doble

Como ya se ha descrito la descompresión doble lleva asociadas dos descompresiones. La primera etapa de descompresión es exactamente igual que la descrita anteriormente con unas pequeñas modificaciones.

En primer caso la primera etapa no descomprime el valor de dato sino que descomprime el valor de contador necesario para la segunda etapa por lo que recibirá un valor de contador y otro valor de cuantas veces se repite dicho contador. Por otra parte como el Microblaze envía los datos a la segunda etapa de uno en uno, es decir un único dato de 12 o 10 bits son enviados en un mensaje de 32 bits, entonces el proceso "Buffer" de la primera descompresión también enviará un único contador por cada mensaje de 32 bits (hay que tener en cuenta que la información a enviar son de 16 bits ya que **se trata de contadores y no de datos**). Por lo tanto el proceso "Buffer" se simplifica, cada vez que recibe un contador lo envía y en este caso da igual que se procese el último contador ya que no se almacenan varios contadores en el mismo buffer, pero si es necesaria esa señal ya que se transmite a la segunda etapa de descompresión (ya que como en la descompresión simple, ahí sí que se trata de datos por lo que se enviaran varios datos en el mismo mensaje). Por lo demás es totalmente igual a la descompresión simple.

La segunda etapa de descompresión también cuenta con una máquina de estados "MDE" y un proceso "Buffer".

La "MDE" en este caso no recibe el número de datos que se van a procesar, ya que como son los obtenidos después de una primera descompresión no se sabe cuántos datos van a mandarse. Por lo tanto en primer lugar se recibe el valor de contador proveniente del coprocesador de la primera descompresión, y en segundo lugar se recibe el valor del dato enviado por Microblaze. Estos dos valores se procesan de igual manera a la descrita en la descompresión simple con la variación que aquí no se utiliza ningún contador ya que no se sabe cuántos datos hay que procesar. Cada vez que se procesa un dato se manda al proceso "Buffer" hasta que el contador es cero y se hace una nueva recepción de datos. En el momento que la primera etapa manda el último dato, también activa una señal que al ser recibida, la "MDE" activa también una señal interna que posteriormente procesará "Buffer". Una vez se ha recibido esa señal y se ha procesado el último dato, ya no hay que hacer más recepciones de datos.

El proceso "Buffer" es exactamente igual que el descrito con anterioridad ya que se tratan de datos, por lo que irá almacenando datos en el buffer de salida según lleguen desde la "MDE" y cuando dicho buffer esté lleno serán enviados. En el caso de tratarse del último dato, como ya se ha dicho el coprocesador de la primera descompresión envía una señal a la MDE del coprocesador de la segunda descompresión que a su vez activa una señal interna que en caso de estar activada, el proceso "Buffer" enviará el buffer de salida al Microblaze aunque no se haya llenado. El funcionamiento es el mismo lo único que cambia es que en la descompresión simple **se sabe cuántos datos se van a procesar** y con la diferencia del número de datos a procesar y número de datos procesados, sabe cuál es el último dato y en el caso de la segunda descompresión directamente **al recibir el dato se avisa con una señal** de comunicación que se trata del último dato.

Por lo tanto se distinguen tres bloques, el primero el Microblaze, el segundo, los tres primeros coprocesadores correspondientes a la primera descompresión y por último los tres segundos coprocesadores correspondientes a la segunda descompresión. El Microblaze se comunica con todos los coprocesadores, la primera etapa recibe información de Microblaze y manda información a la segunda etapa, y la segunda etapa recibe información de Microblaze y de la primera etapa y manda información de vuelta al Microblaze.

4.3. DESCRIPCIÓN DETALLADA DE LOS ALGORITMOS HARDWARE

En este apartado se va a hacer una explicación mucho más detallada de las conexiones entre los distintos módulos, las funciones que cumple cada uno de éstos y una descripción del funcionamiento del código VHDL.

Antes de entrar en el diseño se va a explicar brevemente en qué consiste el Microblaze y cómo funciona la comunicación FSL. Estos dos conceptos ya se introdujeron detalladamente en la memoria del proyecto de compresión por lo que no se va a entrar en detalle, solamente se va a realizar una explicación sencilla para poder comprender como funciona el diseño (si se quieren más detalles consultar la memoria del otro proyecto).

Microblaze es un procesador de 32 bits desarrollado por Xilinx. La arquitectura del procesador Microblaze cuenta con un amplio conjunto de instrucciones optimizadas. Microblaze tiene total flexibilidad para seleccionar la combinación de características de memoria periférica, para tener un mayor rendimiento, una frecuencia mejor o un coste menor de recursos. Cuenta con buses de 32 bits separados para acceso a datos e instrucciones y permite el acceso a estos recursos a través de memoria cache. Microblaze tiene un sistema de interconexión versátil para soportar una gran variedad de aplicaciones embebidas. El bus primario de entradas y salidas de Microblaze (CoreConnect PLB Bus), es un bus de transacción basado en un sistema de memoria mapeado con capacidad maestro/esclavo (que es la que se ha utilizado en este proyecto). Los coprocesadores definidos por el usuario son comunicados a través de una conexión dedicada tipo FIFO, llamada FSL (Fast Simplex Link) [Micro-MB].

FSL es un canal de comunicación punto a punto unidireccional utilizado para realizar una comunicación rápida entre dos elementos de diseño en una FPGA. La interfaz del FSL está disponible en el procesador Microblaze de Xilinx. Las interfaces son utilizadas para transferir los datos hacia y desde el banco de registros en el procesador hardware que se ejecuta en la FPGA [FSL]. El diagrama de bloques de la comunicación FSL es el siguiente:

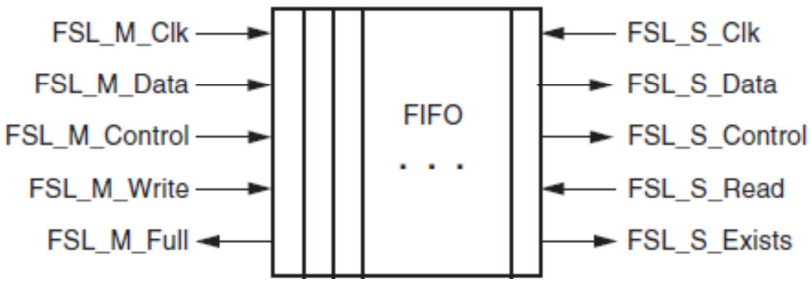


Figura 15: Diagrama de bloques del protocolo de comunicación FSL

Los periféricos del FSL pueden ser creados como maestro o esclavo del bus. Un periférico conectado a los puertos del maestro de un bus FSL envía señales de datos y control al FSL. Un periférico conectado a los puertos del esclavo de un bus FSL recibe señales de datos y control del FSL. Dichas señales son las siguientes:

- **FSL_M_Clk:** Señal de un bit. Señal de entrada que proporciona el reloj a la interfaz maestro del bus FSL.
- **FSL_M_Data:** Señal de 32 bits. Señal de entrada de datos a la interfaz maestro del bus FSL.

- **FSL_M_Control:** Señal de un bit. Señal de entrada de control que se transmite junto con los datos cada flanco de reloj. Esta señal en nuestro diseño no es necesaria por lo que se pondrá a uno cuando se mande el último dato de la primera descompresión a la segunda, así la segunda etapa sabe que es el último dato que tiene que procesar.
- **FSL_M_Write:** Señal de un bit. Señal de entrada que controla la habilitación de escritura de la interfaz maestro del FIFO.
- **FSL_M_Full:** Señal de un bit. Señal de salida la interfaz maestro de FIFO que indica que el FIFO está lleno.
- **FSL_S_Clk:** Señal de un bit. Señal de entrada que proporciona el reloj a la interfaz esclavo del bus FSL.
- **FSL_S_Data:** Señal de 32 bits. Señal de salida de datos de la interfaz esclavo del bus FSL.
- **FSL_S_Control:** Señal de un bit. Señal de salida de control que se transmite junto con los datos cada flanco de reloj. Esta señal es la que recibirá el coprocesador de la segunda descompresión que le indicará que va a recibir el último dato.
- **FSL_S_Read:** Señal de un bit. Señal de entrada que controla la lectura de la interfaz esclavo del FIFO.
- **FSL_S_Exists:** Señal de un bit. Señal de salida del interfaz esclavo que indica que el FIFO contiene datos válidos.

La comunicación FSL ha sido la elegida para comunicar el Microblaze con el diseño y viceversa y también para interconectar los coprocesadores entre sí. El diseño consta de quince FSL´s con las siguientes funciones:

Nombre FSL	Origen	Destino	Información
fsl_0	Microblaze	Desc_primario_1	Envía el valor del número de datos que se van a enviar.
fsl_1	Microblaze	Desc_primario_1	En primer lugar envía el valor del contador de contador y luego el valor del dato de contador, y así sucesivamente. Los datos que envía son de 16 bits.
fsl_2	Microblaze	Desc_secundario_1	Envía el valor del dato de 12 bits.
fsl_3	Desc_primario_1	Desc_secundario_1	Envía el valor del contador reconstruido después de la primera descompresión. Los datos que envía son de 16 bits.
fsl_4	Desc_secundario_1	Microblaze	Envía el dato reconstruido después de la segunda descompresión. Los datos que envía son de 32 bits aunque cada dato son 12 bits.
fsl_5	Microblaze	Desc_primario_2	Envía el valor del número de datos que se van a enviar.
fsl_6	Microblaze	Desc_primario_2	En primer lugar envía el valor del contador de contador y luego el valor del dato de contador, y así sucesivamente. Los datos que envía son de 16 bits.
fsl_7	Microblaze	Desc_secundario_2	Envía el valor del dato de 10 bits.

fsl_8	Desc_primario_2	Desc_secundario_2	Envía el valor del contador reconstruido después de la primera descompresión. Los datos que envía son de 16 bits.
fsl_9	Desc_secundario_2	Microblaze	Envía el dato reconstruido después de la segunda descompresión. Los datos que envía son de 32 bits aunque cada dato son 10 bits.
fsl_10	Microblaze	Desc_primario_3	Envía el valor del número de datos que se van a enviar.
fsl_11	Microblaze	Desc_primario_3	En primer lugar envía el valor del contador de contador y luego el valor del dato de contador, y así sucesivamente. Los datos que envía son de 16 bits.
fsl_12	Microblaze	Desc_secundario_3	Envía el valor del dato de 10 bits.
fsl_13	Desc_primario_3	Desc_secundario_3	Envía el valor del contador reconstruido después de la primera descompresión. Los datos que envía son de 16 bits.
fsl_14	Desc_secundario_3	Microblaze	Envía el dato reconstruido después de la segunda descompresión. Los datos que envía son de 32 bits aunque cada dato son 10 bits.

Tabla 8: Listado de señales de comunicación FSL descompresor doble.

También se ha implementado un segundo diseño, el esquema de la arquitectura es el de la Figura 12 y no muestra ninguna diferencia entre el anterior diseño ya que utiliza el mismo número de FSLs en el que se intenta realizar un mayor aprovechamiento de los FSLs ya que fsl_0, fsl_5 y fsl_10 solo envían una vez el número de datos que se van a descomprimir. Este segundo diseño ha cambiado en que estos tres buses mencionados anteriormente la primera vez mandaran el número de datos que se van a descomprimir y a partir de entonces se encargará de enviar el valor del contador de contador a los coprocesadores de la primera descompresión. Por lo tanto fsl_1, fsl_6 y fsl_11 sólo enviarán el valor del dato de contador a los mismos coprocesadores.

Como ya se ha mencionado anteriormente los seis coprocesadores se pueden dividir en tres dobles descompresores que trabajan en paralelo, es decir, el Desc_primario_1 y Desc_secundario_1, forman un único descompresor doble, y lo mismo pasa con los restantes. Ya que la única diferencia que existe entre ellos es el tamaño de los campos, se va a proceder a explicar uno de ellos y el resto son iguales excepto por dichos campos.

En primer lugar se va a realizar una enumeración con una breve explicación de los puertos de entrada y salida que tienen cada coprocesador. La comunicación FSL impone ciertos puertos, no todos son utilizados por lo que se indicará.

- **Entradas y salidas de Desc_primario_1:**

- **FSL_Clk:** 1 bit de entrada de reloj. Dependiendo de la FPGA utilizada será 50MHz o 100 MHz.
- **FSL_Rst:** 1 bit de entrada de Reset por nivel alto (se puede configurar a nivel bajo).
- **FSL_S_Clk:** 1 bit de entrada de reloj. El reloj del master y esclavo pueden ser distintos pero en este diseño se han utilizado las mismas. Dependiendo de la FPGA utilizada será 50 MHz o 100MHz.
- **FSL_S_Read:** 1 bit de salida al Microblaze. Se activa cuando se quiere leer el número de datos que se van a descomprimir (así el master sabe cuándo tiene que cambiar su dato de salida).
- **FSL_S_Data:** 32 bits de entrada enviados desde el Microblaze que contienen la información del número de datos que se van a descomprimir (en caso del segundo diseño la primera vez contiene ese valor y el resto de veces contendrá el valor del contador de contador).
- **FSL_S_Control:** 1 bit de entrada desde el Microblaze. Es un bit de control que se envía junto a FSL_S_Data que en este caso no se utiliza.
- **FSL_S_Exists:** 1 bit de entrada desde el Microblaze. Cuando está activa indica que el valor del número de datos que se van a procesar está listo en el FSL de entrada (en caso del segundo diseño la primera vez contiene ese valor y el resto de veces contendrá el valor del contador de contador).
- **FSL1_S_Clk:** 1 bit de entrada de reloj. Dependiendo de la FPGA utilizada será 50 MHz o 100MHz.
- **FSL1_S_Read:** 1 bit de salida al Microblaze. Se activa cuando se quiere leer el valor de contador de contador o dato de contador (así el master sabe cuándo tiene que cambiar su dato de salida).
- **FSL1_S_Data:** 32 bits de entrada enviados desde el Microblaze que contienen la información contador de contador o dato de contador (en caso del segundo diseño solo contiene el valor del dato de contador).
- **FSL1_S_Control:** 1 bit de entrada desde el Microblaze. Es un bit de control que se envía junto a FSL_S_Data que en este caso no se utiliza.
- **FSL1_S_Exists:** 1 bit de entrada desde el Microblaze. Cuando está activa indica que el valor de contador de contador o dato de contador están listo en el FSL de entrada (en caso del segundo diseño solo contiene el valor del dato de contador).
- **FSL_M_Clk:** 1 bit de entrada de reloj. Dependiendo de la FPGA utilizada será 50 MHz o 100MHz.
- **FSL_M_Write:** 1 bit de salida enviado a Desc_secundario_1. Cuando se activa se envía el contador de la primera descompresión que esté almacenado en FSL_M_Data**.**
- **FSL_M_Data:** 32 bits de salida enviados a Desc_secundario_1. Contiene el valor del contador de la primera descompresión.
- **FSL_M_Control:** 1 bit de salida enviado a Desc_secundario_1. Es un bit de control que se envía junto con FSL_M_Data que cuando esté activo indicará que el contador que se manda es el último que hay que procesar.
- **FSL_M_Full:** 1 bit de entrada desde Desc_secundario_1. Cuando está a nivel alto indica que el FIFO de la comunicación FSL está lleno por lo que no se pueden enviar más datos hasta que se vacíe y la señal se ponga a nivel bajo.

A parte de las señales de entrada y salida descritas anteriormente existen señales internas solamente utilizadas dentro de cada coprocesador necesarias para poder implementar el algoritmo de descompresión. Las correspondientes al segundo coprocesador son las siguientes:

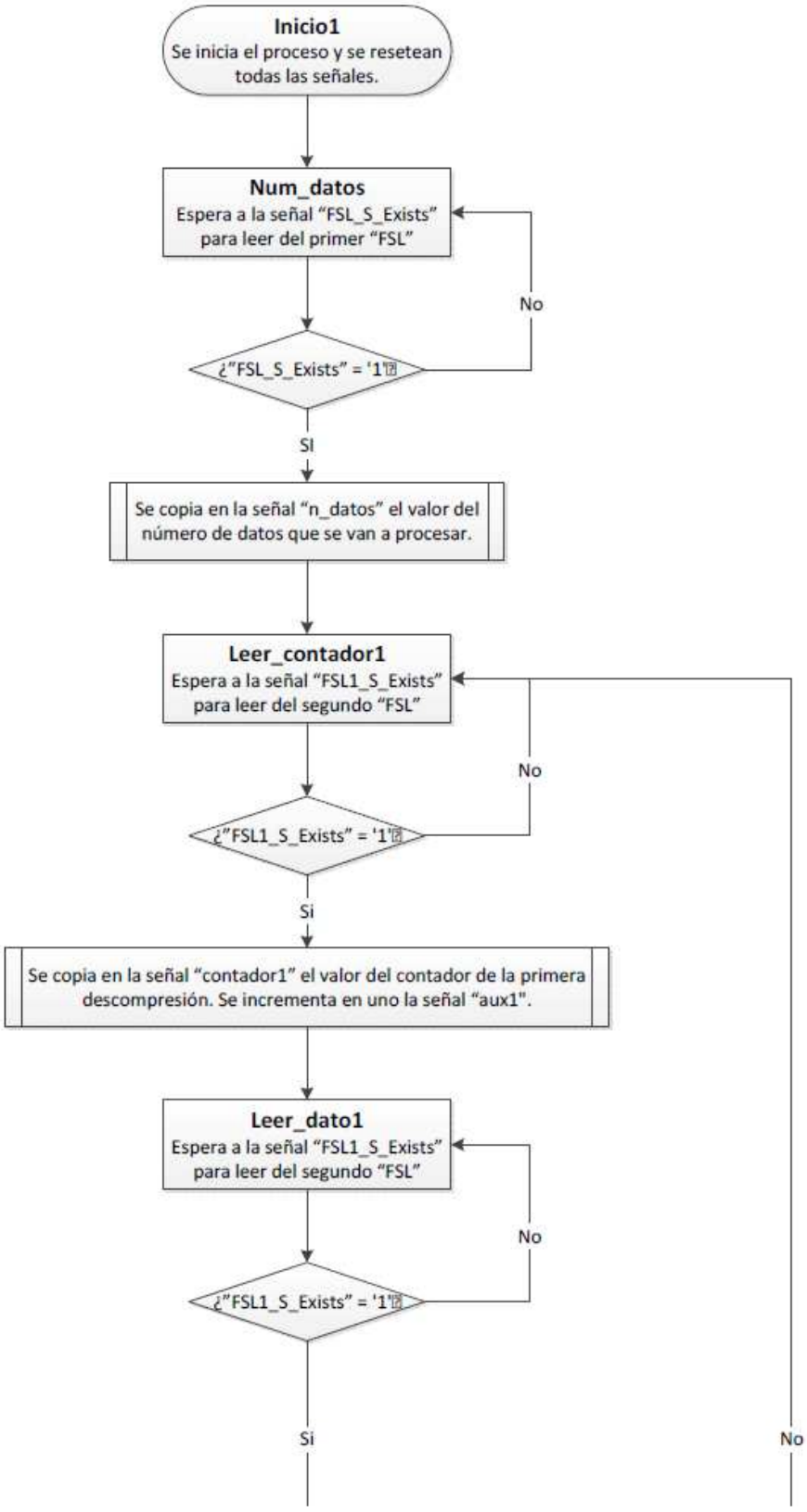
- **Señales internas de Desc_primario_1**

- **env_buff1:** señal de 1 bit. Se pondrá a nivel alto cuando se desee enviar el contador ubicado en FSL_M_Data, el resto del tiempo estará a nivel bajo.
- **n_datos:** señal de 32 bits. En esta señal se almacena el valor de número de datos que se van a descomprimir enviado por Microblaze para posteriormente saber cuándo se está procesando el último dato.
- **contador1:** señal de 32 bits. En esta señal se almacena el valor de contador de contador enviado por Microblaze para poder hacer la descompresión.
- **registro1:** señal de 32 bits. En esta señal se almacena el valor de dato de contador enviado por Microblaze para poder realizar la descompresión.
- **reg1:** señal de 16 bits. En esta señal se almacena los primeros 16 bits de la señal registro1.
- **cont1:** señal de 17 bits. En esta señal se almacena los primeros 16 bits de la señal contador uno. Tiene un bit extra ya que se hacen diversas operaciones con esta señal, en caso de tener todos los bits a nivel alto podría desbordarse y el diseño no funcionaría correctamente.
- **aux1:** señal de 17 bits. En esta señal se almacena el valor del número de datos que se han procesado.
- **buff1:** señal de 44 bits. En esta señal se copia el resultado de la descompresión que posteriormente se enviará al coprocesador de la segunda descompresión.
- **estado_actual1:** es una variable de tipo estado que toma distintos valores según en qué parte de la máquina de estados esté. Puede tomar los siguientes valores: Inicio1, Num_datos, Leer_contador1, Leer_dato1, Procesar_cont1, Procesar_reg1, Escribir1_1 y Comparar1).

Una vez enumeradas tanto las señales de entrada y salida del coprocesador correspondiente a la primera descompresión, y las señales internas de dicho coprocesador, se va a proceder a explicar el funcionamiento interno del algoritmo de descompresión. En primer lugar se realizará una descripción detallada del funcionamiento correspondiente a sólo la primera etapa (es igual que la descompresión simple) y más adelante se enumerarán las señales de entrada y salida, y también las señales internas de la parte correspondiente a la segunda descompresión, ya que son totalmente independientes.

Ha de recordarse que la primera etapa de descompresión se encarga de descomprimir los contadores necesarios para la segunda descompresión, por lo que los datos de entrada serán contador de contador y dato de contador, aparte del número de datos que se van a procesar.

A continuación se muestra el diagrama de flujos de la máquina de estados principal de Desc_primario_1, correspondiente a la primera etapa de descompresión:



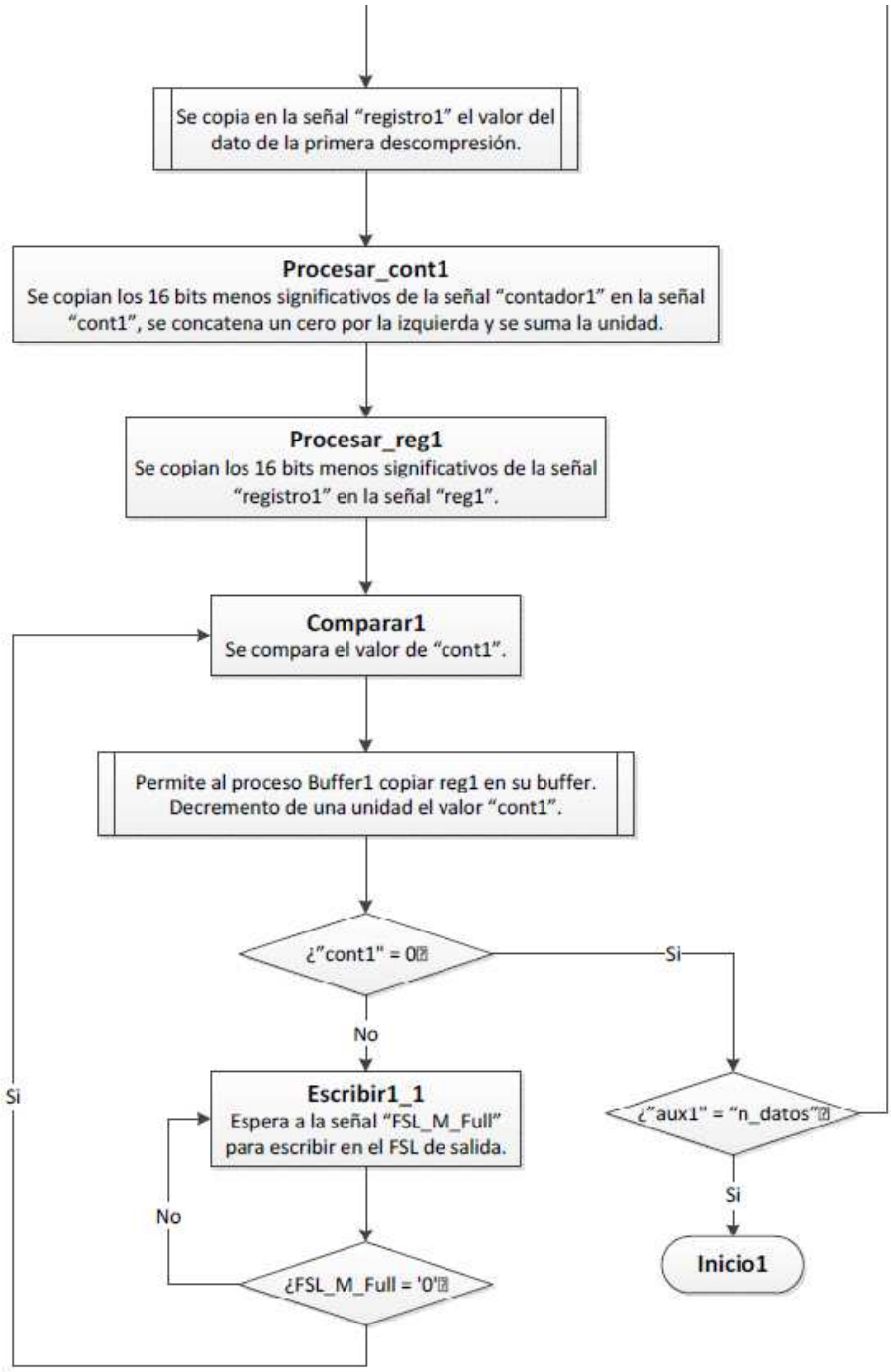


Figura 16: Diagrama de flujo de la máquina de estados de Desc_primario_1.

Esta máquina de estados ha sido etiquetada con el nombre MDE. Por defecto empieza en el estado **Inicio1** donde se inicializan a cero todas las señales. Una vez hecha dicha inicialización se pasa al siguiente estado.

El segundo estado es **Num_datos**. La MDE permanece en este estado siempre que la señal FSL_S_Exists este a nivel bajo, eso quiere decir que no hay ningún dato entrante válido por el bus de comunicación fsl_0. Una vez dicha señal se pone a nivel alto, indicando que ya hay un dato que puede ser leído, se copia en la señal n_datos el valor del número de datos que se van a procesar proveniente de FSL_S_Data. Con lo cual la MDE ya sabe cuántos contadores va a tener que descomprimir. Una vez se ha leído dicho valor se pasa al siguiente estado.

En tercer lugar está el estado **Leer_contador1**. De igual manera la MDE permanece esperando en este estado hasta que el valor de la señal FSL1_S_Exists esté a nivel alto, indicando que hay un valor de contador de contador listo para ser transmitido, dicho valor procede del bus de comunicación fsl_1. Una vez se dan las condiciones anteriores, se copia el valor de FSL1_S_Data (contiene el valor de contador de contador) en la señal contador1. En este estado también se incrementa en una unidad el valor de aux1 que contiene la información de cuántos datos se han procesado, es decir, cada vez que se lee un valor de contador de contador se considera que se empieza a procesar un dato distinto del anterior. En el segundo diseño que se ha implementado este estado sólo varia en que el dato lo lee de un FSL distinto, en ese caso esperaría a que la señal FSL_S_Exists estuviera a nivel alto y copiaría los valores desde FSL_S_Data, es decir del bus fsl_0, por lo demás el resto del diseño es igual. Después de esto se pasa al siguiente estado.

El cuarto estado es **Leer_dato1**. Este estado es muy parecido al anterior ya que recibe un dato desde Microblaze, esta vez por el bus de comunicación fsl_1. La MDE se mantiene en dicho estado hasta que la señal FSL1_S_Exists pasa a estar en nivel alto. Una vez se ha dado esta situación se copia en la señal interna registro1 el valor de dato de contador transmitido por FSL1_S_Data. Una vez que el valor del dato de contador ha sido almacenado la MDE pasa al siguiente estado.

En quinto lugar está el estado **Procesar_cont1**. En este estado se realiza una copia de los 16 bits menos significativos de la señal contador1 (los bits que llevan la información de contador de contador) en la señal cont1. La señal interna cont1 es de 17 bits por lo que para que tengan el mismo tamaño, esos 16 bits que se copian de la señal contador1 se concatenan con un cero lógico por la izquierda. A continuación ese valor copiado en cont1 y es incrementado en una unidad. Esto se hace ya que si el dato de contador se repite una vez (contador de contador es uno), entonces quiere decir que dato de contador hay que enviarlo al coprocesador de la segunda descompresión dos veces, por eso al valor de contador de contador se le suma una unidad. La señal cont1 tiene un tamaño mayor al contador (la señal es de 17 bits y el contador de contador de 16 bits) ya que en caso que el contador tenga todos sus bits a nivel alto, al sumarle la unidad, la señal desbordaría y el descompresor no funcionaría correctamente, por eso se le añade un bit extra. Una vez realizada esta operación la MDE pasa al siguiente estado.

El sexto estado es **Procesar_reg1**. De igual manera que en el estado anterior hay que copiar la información de dato de contador en una señal interna. En este caso dicha señal interna es reg1, donde se copian los 16 bits menos significativos (donde está la información de dato de contador) de la señal interna registro1. Como en este caso no se hace ninguna operación, la señal reg1 también es de 16 bits. Una vez la información esta guardada en reg1 la MDE pasa al siguiente estado.

En séptimo lugar está el estado **Comparar1**. Este estado es el que se encarga de evaluar las condiciones en la que se encuentra el proceso de descompresión y según cuales sean éstas continuar el proceso de una forma u otra. Se pueden tener tres posibilidades distintas, que el dato de contador que se acaba de procesar se repita, que el dato de contador que se acaba de procesar no se repita pero que aún queden datos por descomprimir, y que el dato de contador que se acaba de procesar no se repita y que sea el último dato que hay que descomprimir. Cuando se dice que el dato de contador no se repita, se quiere decir que, o bien el contador de contador es cero o bien que si se repite 3 veces (habrá que mandarlo 4 veces ya que se cuentan repeticiones) ya se haya enviado tres veces y ésta sea la última. Como ya se ha dicho con anterioridad la señal interna que tiene la información de contador de contador (cuantas veces se repite dato de contador) es cont1, como ya se verá en el siguiente estado cada vez que se haga una escritura de datos el valor de esta señal será disminuido en una unidad hasta que dicho valor llegue a cero lo que significará que ya se han realizado tantos envíos como repeticiones tenía. Por lo tanto, las tres opciones son las siguientes, en caso que cont1 valga cero (ya se han procesado todas las veces necesarias) existen dos opciones, que aux1 (número de datos de contador que se han procesado) y n_datos (número de datos de contador que se van a procesar) sean iguales, lo que quiere decir que ya se han procesado todos los datos de contador, entonces el siguiente estado de la MDE es **Inicio1**, o bien que aux1 sea menor que n_datos, lo que quiere decir que aún no se han procesado todos los datos de contador por lo que hay más valores de entrada por FSL, en tal caso el siguiente estado es **Leer_contador1**, por el contrario está la opción que cont1 tenga un valor mayor que cero lo que indicará que aún no se ha procesado por completo ese dato de contador (quedan repeticiones por hacer), en tal caso el siguiente estado es Escribir1_1.

Por último, en octavo lugar, está el estado **Escribir1_1**. La MDE permanece en este estado siempre y cuando la señal FSL_M_Full esté a nivel alto, indicando que el FSL de salida está lleno. Cuando dicha señal pase a estar a nivel bajo indicando que ya se puede utilizar el FSL de salida, se disminuirá en una unidad el valor de la señal cont1 como se ha dicho anteriormente, indicando que se ha escrito el valor reg1 en el bus FSL de salida por lo que es una repetición que se ha enviado. Hay que tener en cuenta que en la MDE no se hace la escritura en el FSL, se crea un estado para ello, pero la escritura se realiza en un proceso independiente, el cuál cuando la MDE esté en el estado Escribir1_1, escribirá el valor de reg1 en el FSL de salida, como se verá a continuación. Es decir en la MDE se da la orden a otro proceso de que escriba y en la MDE se contabiliza dicha escritura.

Este es el funcionamiento detallado de la MDE correspondiente al coprocesador Desc_primario_1, véase Figura 12 responsable de hacer la primera etapa de descompresión del primer campo. Se recuerda que el diseño consta de tres campos distintos, pero como ya se ha dicho todos los contadores de contadores y datos de contadores, al tratarse de contadores, son de 16 bits por lo que los tres coprocesadores de la primera etapa de descompresión (Desc_primario_1, Desc_primario_2 y Desc_primario_3) son exactamente iguales.

Una vez se ha realizado la explicación de la máquina de estados principal se va a proceder a explicar el funcionamiento del proceso Buffer1 que es el encargado de escribir en el FSL de salida hacia el coprocesador de la segunda descompresión.

A continuación se muestra el diagrama de flujos del proceso de escritura Buffer1 de Desc_primario_1, correspondiente a la primera etapa de descompresión:

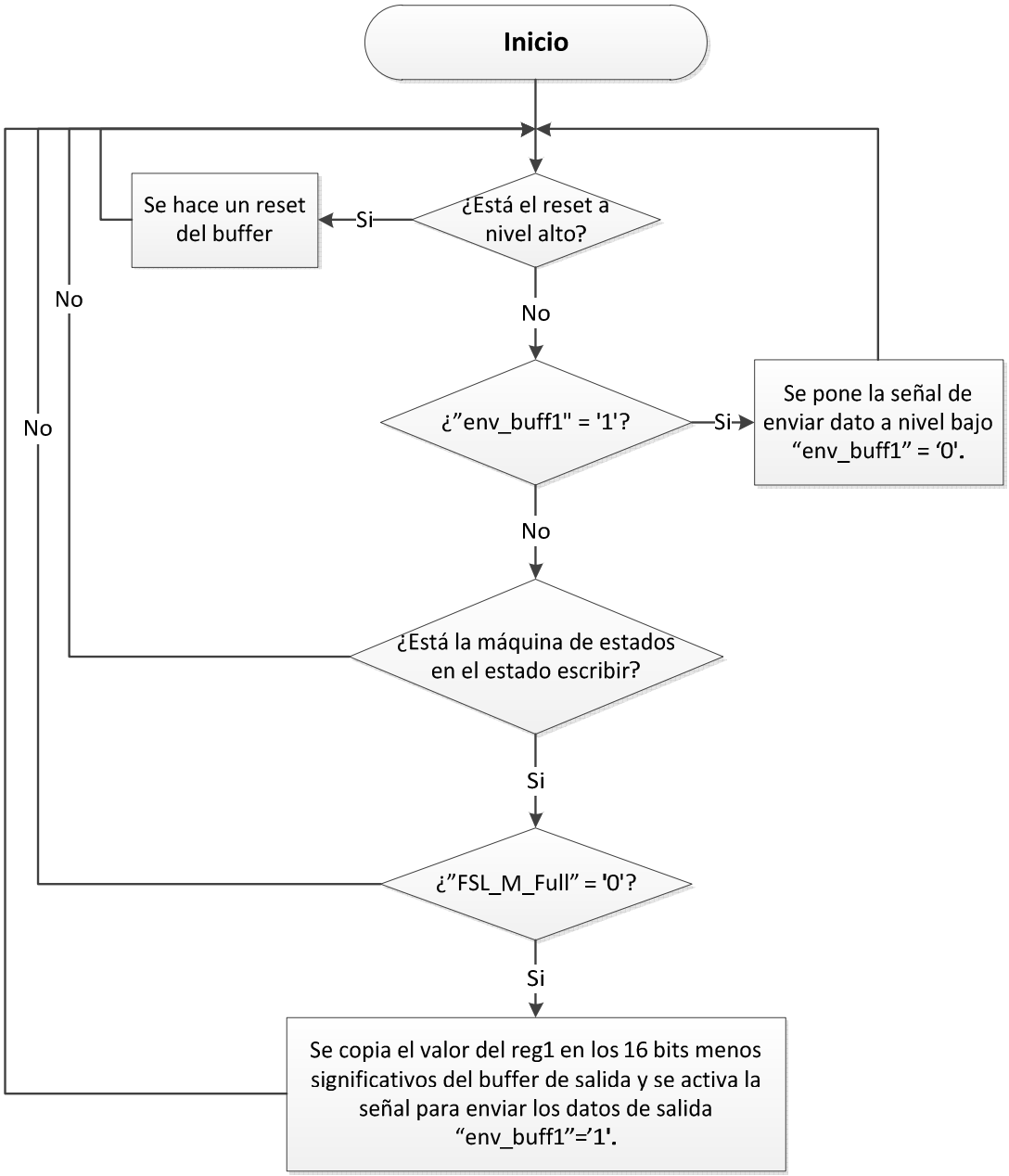


Figura 17: Diagrama de flujos del proceso Buffer1 de Desc_primario_1.

Buffer1 es un proceso que funciona según unas condiciones, depende de en qué estado se encuentre MDE, el valor de alguna señal interna y el valor de una señal de comunicación FSL.

En primer lugar si hay un reset se inicializa a cero la señal interna buff1 (la encargada de almacenar el valor del contador obtenido tras la primera descompresión.

En caso que el reset no esté a nivel alto se comprueba el valor de la señal interna env_buff1 (se pone a nivel alto cuando se quieren enviar los datos por el FSL de salida) en caso que esté a nivel alto quiere decir que se acaba de hacer una transmisión por lo que habrá que su valor cambiará a un cero lógico. Si dicha señal estaba a nivel alto, se sale del bucle y vuelve a empezar comprobando si el reset está a nivel alto, en caso contrario pasa a comprobar en qué estado se encuentra la MDE, si la MDE está en **Escribir1_1** el proceso continuaría, pero lo normal es que no lo esté por lo que el proceso haría un bucle haciendo las dos comprobaciones anteriormente descritas hasta que la máquina de estados principal llegue al estado de escribir.

Una vez la MDE está en dicho estado se comprobará si el bus FSL de salida está disponible o no, en caso de estar lleno, FSL_M_Full es cero, se saldrá del bucle y volverá a iniciarlo pero como la MDE tiene la misma condición para continuar el proceso las condiciones serán las mismas hasta que el puerto FSL de salida esté disponible para escribir por lo que el proceso Buffer1 podrá continuar.

Cuando Buffer1 haya comprobado que el bus FSL de salida está disponible copiará el valor de la señal interna reg1 (como se ha visto antes es el valor de dato de contador) en los 16 bits menos significativos de otra señal interna llamada buff1. Una vez hecho esto se pondrá env_buff1 a nivel alto indicando que hay un dato en el FSL de salida listo para enviarse.

Como se ha visto anteriormente **el canal de comunicación FSL impone unos puertos determinados** para realizar dicha comunicación y en lo descrito hasta ahora no se han utilizado dichos puertos, eso es porque se han utilizado señales internas para realizar la descompresión en procesos y asignar dichas señales fuera de los procesos, es decir igualamos el valor de los puertos FSL a valores de señales internas, por lo que al cambiar el valor de las señales internas cambiará instantáneamente el valor de los puertos de comunicación FSL.

Dichas asignaciones son las siguientes:

- **FSL_S_Read = '1':** cuando la MDE está en el estado Num_datos, es decir cuando está en el estado de leer el número de datos que se van a procesar se pide al FSL que mande el dato para poder leerlo. En cualquier otro estado la señal vale '0'. En caso del segundo diseño que se ha implementado esa condición se extiende también cuando la MDE está en el estado Leer_contador1 ya que por este FSL también se lee el valor de contador de contador.
- **FSL1_S_Read = '1':** cuando la MDE está en el estado Leer_contador1 o Leer_dato1, cuando se quiere leer un nuevo contador de contador o dato de contador se pide al FSL que mande dicho dato. En cualquier otro estado dicha señal vale '0'. En el caso del segundo diseño que se ha implementado la condición sólo sería cuando la MDE está en el estado Leer_dato1 ya que por este FSL sólo se leerá el valor de dato de contador.
- **FSL_M_Control = '1':** cuando aux1 tiene el mismo valor que n_datos y cont1 = '0' se han descomprimido hasta la última repetición del último dato, es decir, se han descomprimido todos los datos. Al enviar esta señal junto con el último dato del valor de contador se indica al coprocesador encargado de la segunda descompresión que ese es el último dato que tiene que descomprimir.
- **FSL_M_Write = env_buff1**: el valor que indica cuando se han de enviar los datos por el bus FSL de salida se iguala al valor de una señal interna, es decir, cuando en el proceso Buffer1 se pone a nivel alto env_buff1 a su vez se pone a nivel alto FSL_M_Write, por lo que el valor que esté en FSL_M_Data se enviará.

- **FSL_M_Data = buff1**: el dato que se envía a través del FSL de salida es el valor que hay almacenado en los 32 bits menos significativos de la señal interna buff1, que a su vez sus 16 bits menos significativos son los que contienen la información del valor del contador necesario para la segunda descompresión.

Al igual que la MDE el proceso Buffer1 está en cada uno de los tres coprocesadores correspondientes a la primera etapa de descompresión (Desc_primario_1, Desc_primario_2 y Desc_primario_3) y como se ha dicho anteriormente al tratarse de una descompresión de contadores de 16 bits en los tres campos el proceso Buffer1 será igual.

Una vez que ha sido descrito detalladamente el funcionamiento de la parte correspondiente a la primera descompresión se va a proceder a describir la segunda descompresión.

Al igual que con el coprocesador correspondiente a la primera descompresión en este caso se van a volver a enumerar los puertos utilizados para la comunicación FSL.

- **Entradas y salidas de Desc_secundario_1:**

  - **FSL_Clk:** 1 bit de entrada de reloj. Dependiendo de la FPGA utilizada será 50MHz o 100 MHz.
  - **FSL_Rst:** 1 bit de entrada de Reset por nivel alto (se puede configurar a nivel bajo).
  - **FSL2_S_Clk:** 1 bit de entrada de reloj. El reloj del master y esclavo pueden ser distintos pero en este diseño se han utilizado las mismas. Dependiendo de la FPGA utilizada será 50 MHz o 100MHz.
  - **FSL2_S_Read:** 1 bit de salida a Desc_primario_1. Se activa cuando se quiere leer el valor de contador que se ha obtenido después de la primera descompresión (así el máster sabe cuándo tiene que cambiar su dato de salida).
  - **FSL2_S_Data:** 32 bits de entrada enviados desde Desc_primario_1 que contienen la información del valor de contador obtenido después de la primera descompresión.
  - **FSL2_S_Control:** 1 bit de entrada desde Desc_primario_1. Es un bit de control que se envía junto a FSL_S_Data que en este caso no se utiliza. Cuando este puerto está a nivel alto indica que el dato que se está recibiendo es el último que hay que descomprimir.
  - **FSL2_S_Exists:** 1 bit de entrada desde Desc_primario_1. Cuando está activa indica que el valor de contador está listo en el FSL de entrada.
  - **FSL3_S_Clk:** 1 bit de entrada de reloj. Dependiendo de la FPGA utilizada será 50 MHz o 100MHz.
  - **FSL3_S_Read:** 1 bit de salida al Microblaze. Se activa cuando se quiere leer el valor de dato (así el master sabe cuándo tiene que cambiar su dato de salida).
  - **FSL3_S_Data:** 32 bits de entrada enviados desde el Microblaze que contienen la información dato.
  - **FSL3_S_Control:** 1 bit de entrada desde el Microblaze. Es un bit de control que se envía junto a FSL_S_Data que en este caso no se utiliza.
  - **FSL3_S_Exists:** 1 bit de entrada desde el Microblaze. Cuando está activa indica que el valor dato está listo en el FSL de entrada.
  - **FSL3_M_Clk:** 1 bit de entrada de reloj. Dependiendo de la FPGA utilizada será 50 MHz o 100MHz.

- **FSL3_M_Write:** 1 bit de salida enviado al Microblaze. Cuando se activa se envía el dato obtenido después de la doble descompresión que esté almacenado en FSL_M_Data**.**
- **FSL3_M_Data:** 32 bits de salida enviados al Microblaze. Contiene el valor de dato obtenido después de la doble descompresión.
- **FSL3_M_Control:** 1 bit de salida al Microblaze. Es un bit de control que se envía junto a FSL_S_Data que en este caso no se utiliza.
- **FSL3_M_Full:** 1 bit de entrada desde el Microblaze. Cuando está a nivel alto indica que el FIFO de la comunicación FSL está lleno por lo que no se pueden enviar más datos hasta que se vacíe y la señal se ponga a nivel bajo.

A parte de las señales de entrada y salida descritas anteriormente existen señales internas solamente utilizadas dentro de cada coprocesador necesarias para poder implementar el algoritmo de descompresión. Las correspondientes al segundo coprocesador son las siguientes:

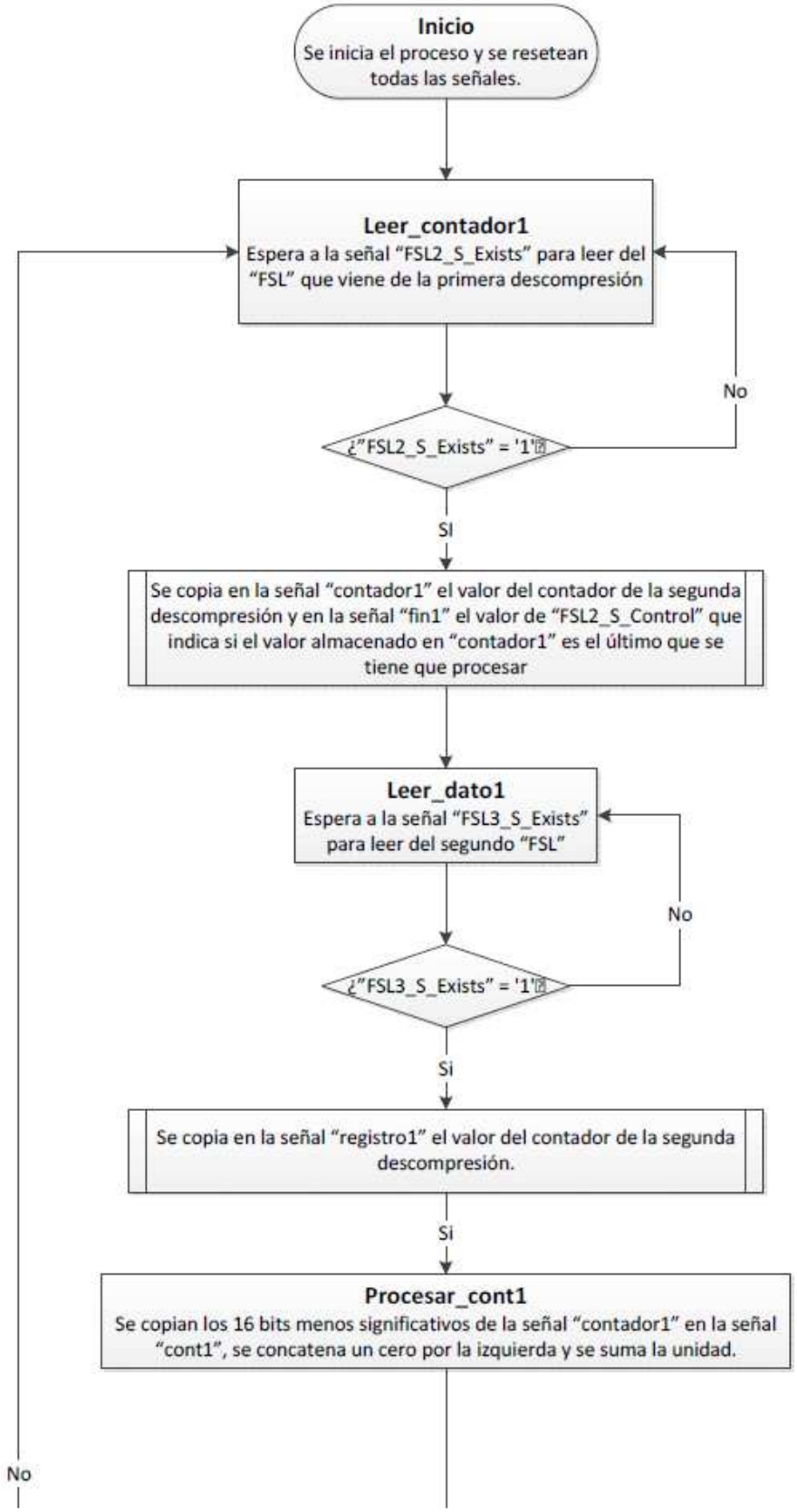
- **Señales internas de Desc_secundario_1**

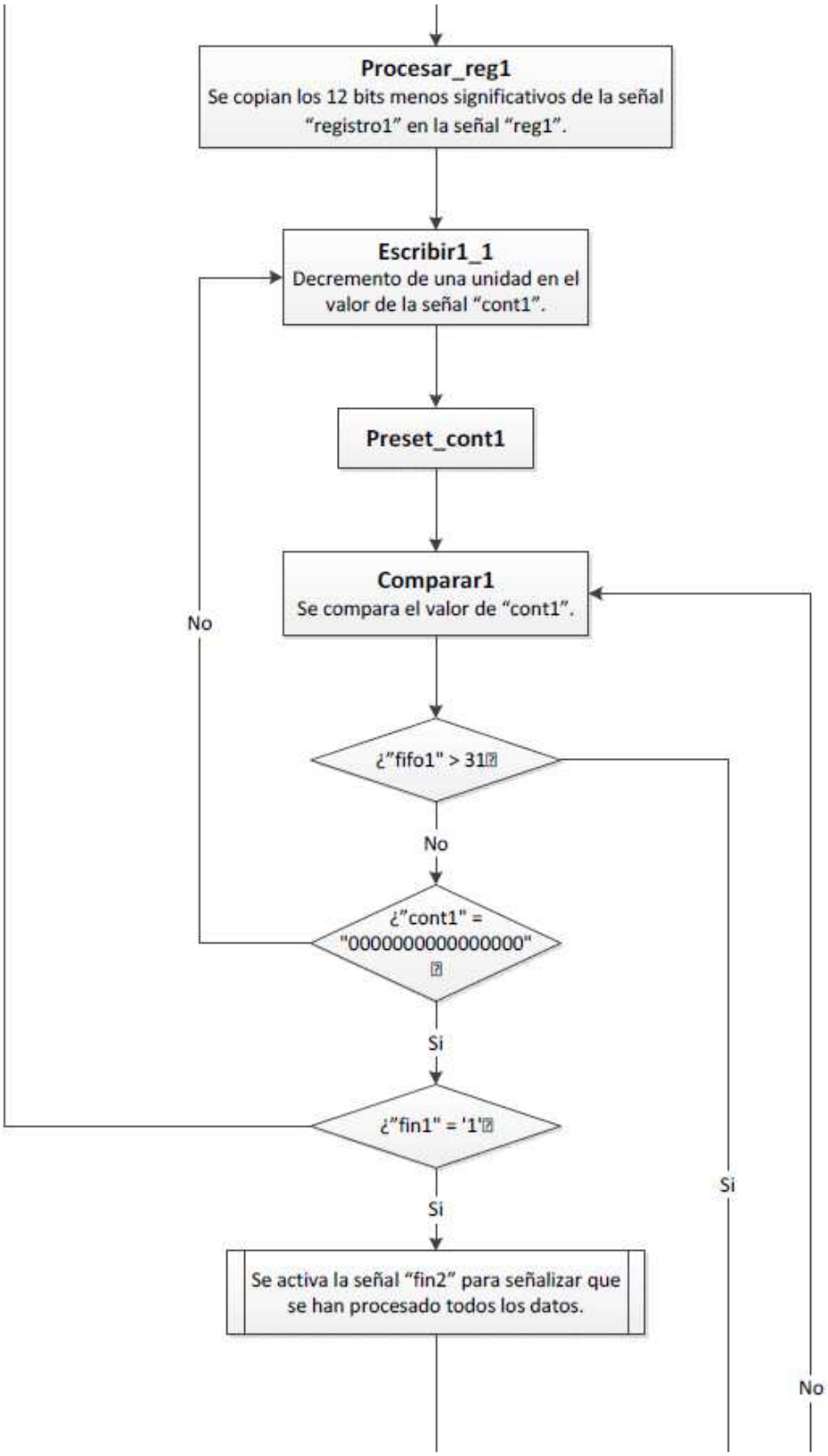
- **env_buff1:** señal de 1 bit. Se pondrá a nivel alto cuando se desee enviar el contador ubicado en FSL3_M_Data, el resto del tiempo estará a nivel bajo.
- **fin1:** señal de 1 bit. Tomará el valor de FSL3_M_Control que se usará para saber cuándo se está descomprimiendo el último dato.
- **fin2:** señal de 1 bit. Se usará para la descompresión del último dato.
- **contador1:** señal de 32 bits. En esta señal se almacena el valor de contador de contador enviado por Microblaze para poder hacer la descompresión.
- **registro1:** señal de 32 bits. En esta señal se almacena el valor de dato de contador enviado por Microblaze para poder realizar la descompresión.
- **reg1:** señal de 12 bits. En esta señal se almacena los primeros 16 bits de la señal registro1.
- **cont1:** señal de 17 bits. En esta señal se almacena los primeros 16 bits de la señal contador uno. Tiene un bit extra ya que se hacen diversas operaciones con esta señal, en caso de tener todos los bits a nivel alto podría desbordarse y el diseño no funcionaría correctamente.
- **buff1:** señal de 44 bits. En esta señal se copia el resultado de la descompresión que posteriormente se enviará al coprocesador de la segunda descompresión.
- **fifo1:** señal entera. Se utiliza para discriminar donde hay que copiar los datos en buff1. La primera vez se escribe en el bit significativo pero la segunda vez hay que empezar a copiar el dato donde terminó el anterior.
- **estado_actual1:** es una variable de tipo estado que toma distintos valores según en qué parte de la máquina de estados esté. Puede tomar los siguientes valores: Inicio1, Leer_contador1, Leer_dato1, Procesar_cont1, Procesar_reg1, Escribir1_1, Preset_cont1, Comparar1, Escribir2_1 y Volver1).

Una vez enumeradas tanto las señales de entrada y salida del coprocesador correspondiente a la segunda descompresión, y las señales internas de dicho coprocesador, se va a proceder a explicar el funcionamiento interno del algoritmo de descompresión de dicha parte.

Dicho coprocesador es el encargado de descomprimir el dato inicial (antes de la compresión) mediante el contador descomprimido por la primera etapa más el dato enviado directamente desde Microblaze.

A continuación se muestra el diagrama de flujos de la máquina de estados principal de Desc_secundario_1, correspondiente a la segunda etapa de descompresión:




Procesar_reg1
Se copian los 12 bits menos significativos de la señal "registro1" en la señal "reg1".
Escribir1_1
Decremento de una unidad en el valor de la señal "cont1".
Preset_cont1
Comparar1
Se compara el valor de "cont1".
No
¿"fifo1" > 31
No
¿"cont1" = "0000000000000000"
Si
¿"fin1" = '1'
Si
Si
Se activa la señal "fin2" para señalizar que se han procesado todos los datos.
No

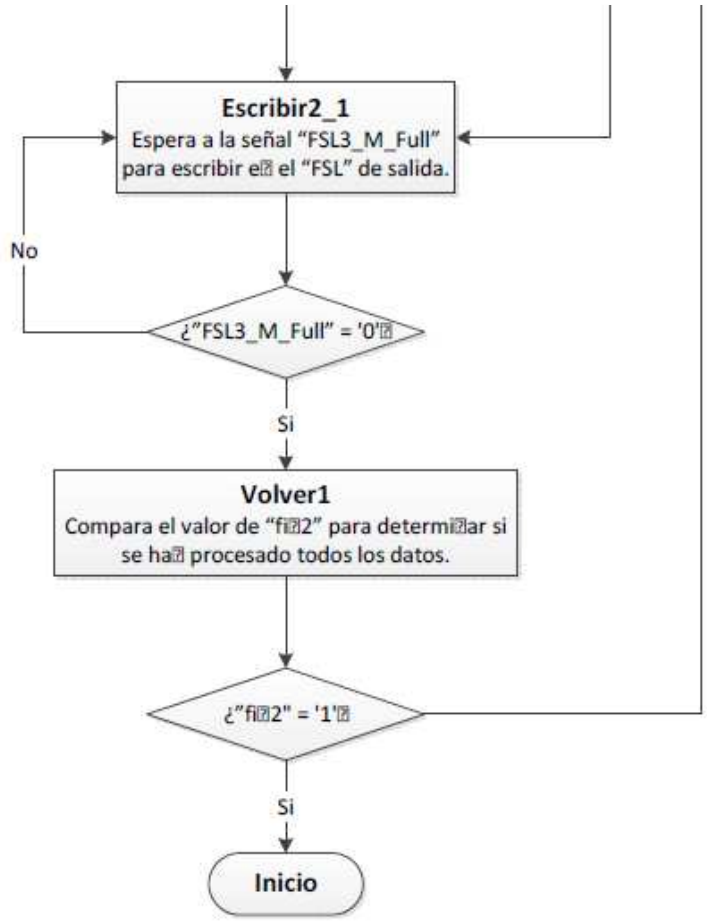


Figura 18: Diagrama de flujo de la máquina de estados de Desc_secundario_1.

Esta máquina de estados ha sido etiquetada con el nombre MDE. Por defecto empieza en el estado **Inicio1** donde se inicializan a cero todas las señales. Una vez hecha dicha inicialización se pasa al siguiente estado.

En segundo lugar está el estado **Leer_contador1**. La MDE permanece esperando en este estado hasta que el valor de la señal FSL2_S_Exists esté a nivel alto, indicando que hay un valor de contador listo para ser transmitido, dicho valor procede del bus de comunicación fsl_3. Una vez se dan las condiciones anteriores, se copia el valor de FSL2_S_Data (contiene el valor de contador) en la señal interna contador1.También en este estado se copia en la señal interna fin1 el valor de FSL2_S_Control, en caso de estar a nivel alto indica que el dato que se acaba de copiar en contador1 es el último que hay que procesar, por lo que, si no se llena el buffer al terminar la descompresión se mandará ocupando los huecos restantes con ceros. Después de esto se pasa al siguiente estado.

El tercer estado es **Leer_dato1**. Este estado es muy parecido al anterior ya que recibe un dato desde Microblaze, esta vez por el bus de comunicación fsl_2. La MDE se mantiene en dicho estado hasta que la señal FSL3_S_Exists pasa a estar en nivel alto. Una vez se ha dado esta situación se copia en la señal interna registro1 el valor de dato recibido por FSL3_S_Data. Una vez que el valor del dato ha sido almacenado la MDE pasa al siguiente estado.

En cuarto lugar está el estado **Procesar_cont1**. En este estado se realiza una copia de los 16 bits menos significativos de la señal contador1 (los bits que llevan la información de contador) en la señal cont1. La señal interna cont1 es de 17 bits por lo que para que tengan el mismo tamaño, esos 16 bits que se copian de la señal contador1 se concatenan con un cero lógico por la izquierda. A continuación ese valor copiado en cont1 y es incrementado en una unidad. Esto se hace ya que si el dato se repite una vez (contador es uno), entonces quiere decir que dato hay que enviarlo de vuelta al Microblaze dos veces, por eso al valor de contador se le suma una unidad. La señal cont1 tiene un tamaño mayor al contador (la señal es de 17 bits y el contador de 16 bits) ya que en caso que el contador tenga todos sus bits a nivel alto, al sumarle la unidad, la señal desbordaría y el descompresor no funcionaría correctamente, por eso se le añade un bit extra. Una vez realizada esta operación la MDE pasa al siguiente estado.

El sexto estado es **Procesar_reg1**. De igual manera que en el estado anterior hay que copiar la información de dato en una señal interna. En este caso dicha señal interna es reg1, donde se copian los 16 bits menos significativos (donde está la información de dato) de la señal interna registro1. Como en este caso no se hace ninguna operación, la señal reg1 también es de 16 bits. Una vez la información esta guardada en reg1 la MDE pasa al siguiente estado.

En séptimo lugar está el estado **Escribir1_1**. Este estado permite al proceso Buffer1 copiar la señal reg1 en el buffer de salida, en el caso de la primera etapa de compresión copiaba el valor en el buffer y lo mandaba, en este caso se va rellenando el buffer de 32 bits con datos de 12 bits, una vez éste esté lleno se enviará. Como se ha procesado un dato se disminuye en una unidad el valor de cont1 que es la señal que lleva el valor de cuántas veces se repite el dato. Una vez que se ha actualizado el valor de dicha señal se pasa al siguiente estado.

El octavo estado es **Preset_cont1**. En la MDE no realiza ninguna operación, ya que su función es crear un estado para que el proceso Buffer pueda realizar una operación. Dicha operación consta de ajustar el valor de la señal fifo1, esas operaciones se explicarán cuando se llegue a la descripción del proceso Buffer. Como en este estado no se realiza ninguna operación en el próximo ciclo de reloj la MDE pasa al siguiente estado.

En noveno lugar está el estado **Comparar1**. Este estado es el que se encarga de evaluar las condiciones en la que se encuentra el proceso de descompresión y según cuales sean éstas continuar el proceso de una forma u otra. Se pueden tener tres posibilidades distintas, que el buffer se haya llenado, que se haya terminado de procesar un dato y éste sea el último, que se haya terminado de procesar un dato pero no sea el último y que no se haya terminado de procesar el dato. El primer caso posible es que la señal fifo1 sea mayor que 31 (fifo1 indica en qué bit de la señal buffer hay que copiar la señal reg1, es como un puntero que indica donde se tiene que empezar a copiar la señal reg1) como el buffer de salida tiene un tamaño limitado a 32 bits, si fifo1 es mayor que 31 ese dato no se podrá enviar por lo que el próximo estado será Escribir2_1 donde se enviará el buffer lleno. La segunda opción es que la señal cont1 sea cero, lo que quiere decir que ya se han procesado todas las repeticiones del dato, en ese caso hay dos opciones, la primera es que ese dato sea el último que hay que descomprimir, que en tal caso se pondría a nivel alto la señal fin2 (para indicar que es el último dato que se procesa) y se enviaría el buffer de salida esté lleno o no, en tal caso el siguiente estado también será Escribir2_1, la segunda opción es que no sea el último dato, es decir que fin1 esté a nivel alto por lo que habrá que recibir el siguiente contador y dato a descomprimir, en tal caso el siguiente estado será Leer_contador1. Por último existe la opción que la señal cont1 no sea cero, es decir, no se han enviado todas las repeticiones del dato almacenado en reg1, en tal caso el siguiente estado será Escribir1_1. Una vez evaluado el caso la MDE continuará por el estado correspondiente, como ya se ha descrito todos los estados excepto una de las opciones se explicará ese estado.

El décimo estado es **Escribir2_1**. Este estado es el que se encarga de indicarle al proceso Buffer1 que mande los por el FSL de salida. A este estado se llega bien porque la señal fifo1 es mayor que 32, es decir ya se ha llenado en buffer de salida o bien porque ya se ha procesado el último dato, por lo que habrá que enviar los datos existentes y los bits libres rellenarlos con ceros. Como todos los estados en los que se escribe en un FSL, la MDE permanecerá en dicho estado siempre que el bus de comunicación esté lleno, es decir, siempre que FSL3_M_Full = '1', una vez que pase a estar a nivel bajo, indicará que el bus de comunicación FSL está listo para transmitir, dicho bus es el fsl_4. Como en todos los casos la MDE no se encarga de la escritura, sino que tiene un estado por el cual indica al proceso Buffer que tiene que escribir, es decir si no existiese ese segundo proceso la MDE esperaría a las condiciones para poder transmitir los datos pero una vez que son las correctas pasa al siguiente estado, y es Buffer1 el que se encarga de mandar esos datos. Una vez se han dado dichas condiciones se pasa al último estado.

En undécimo y último lugar está el estado **Volver1**. En este estado se evaluará si el dato que se acaba de mandar por el FSL de salida era el último que había que descomprimir o si por lo contrario hay más datos que descomprimir. Esta condición se comprueba comparando el valor de la señal fin2, que en caso de ser el último dato a descomprimir, en el estado Comparar uno dicha señal se ponía a nivel alto, en caso contrario permanecerá a nivel bajo. Por lo tanto si fin2 está a nivel alto ya se ha terminado de descomprimir todos los datos por lo que el siguiente estado será Inicio1, y en caso contrario el siguiente estado será Comparar 1 ya que para llegar al estado Volver1 siendo fin2 = '0', ha sido porque fifo1 era mayor que 32 es decir el buffer se había llenado por lo que se vacía y se vuelve a comparar cual es la situación y actuar en consecuencia.

En éste caso los tres coprocesadores encargados de la segunda descompresión (Desc_secundario_1, Desc_secundario_2 y Desc_secundario_3) sí que tienen una pequeña diferencia, el funcionamiento es el mismo pero Desc_secundario_1 descomprime datos de 12 bits, y tanto Desc_secundario_2 como Desc_secundario_3 descomprimen datos de 10 bits, sumando un total de 32 bits (tamaño de la comunicación FSL). Estos valores del tamaño de campo que descomprime cada coprocesador es totalmente configurable. Por lo demás los procesos son idénticos.

Una vez se ha realizado la explicación de la máquina de estados principal se va a proceder a explicar el funcionamiento del proceso Buffer1 que es el encargado de escribir en el FSL de salida de vuelta al Microblaze.

A continuación se muestra el diagrama de flujos del proceso de escritura Buffer1 de Desc_secundario_1, correspondiente a la segunda etapa de descompresión:

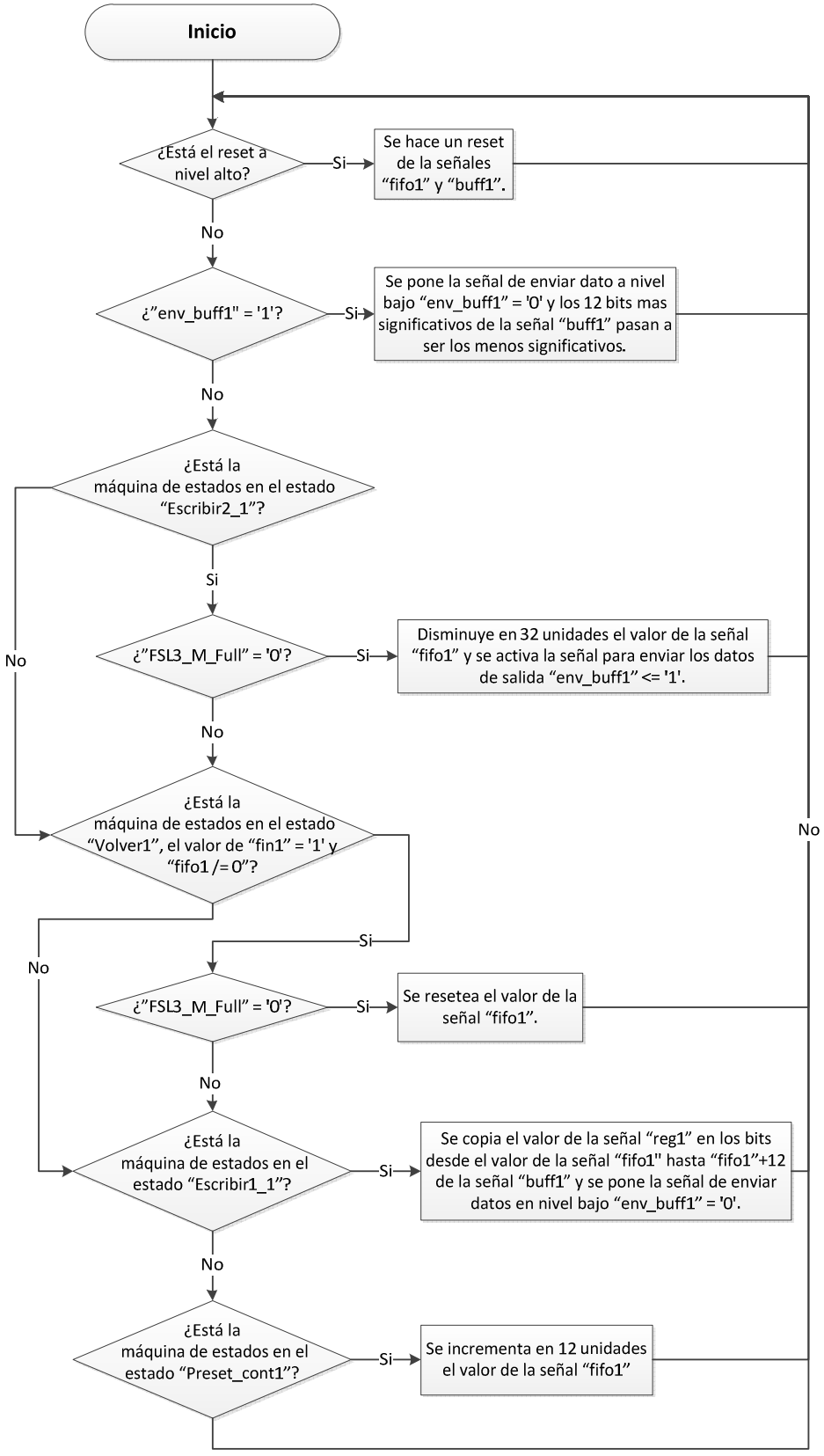


Figura 19: Diagrama de flujos del proceso Buffer1 de Desc_secundario_1.

En el caso de los coprocesadores encargados de la segunda etapa de descompresión, el proceso Buffer1 es algo más complicado que en la primera etapa. Como ya se ha dicho en este caso se irán llenando los 32 bits que se pueden enviar por el bus FSL con datos de 12 bits, una vez se hayan llenado se enviará y se empezara a llenar el siguiente dato de salida.

Por lo tanto el proceso empieza en una **etapa de reposo**. En caso de haber un reset, las señales fifo1 y buff1 se inicializan a cero y el proceso vuelve a estar en reposo, en caso contrario se comprueba el valor de la señal env_buff1, en caso que esté a nivel alto indica que se acaba de hacer una transmisión por lo que se modificará su valor a nivel bajo, como se acaba de hacer una transmisión de datos quiere decir que el buffer estaba lleno y es posible que desbordado, por eso la señal **buff1 es de 44 bits** ya que en el peor de los casos fifo1 valdrá 31 antes de una transmisión, si se copia en buff1 12 bits más se copiarán del bit 32 al 43 ambos incluidos por lo que se transmitirán los primeros 32 bits de la señal buff1 y posteriormente habrá que hacer un desplazamiento en el que el bit 33 pase a ser el bit menos significativo dentro de buff1, aparte en la próxima copia de datos habrá que hacerlo a partir del bit 11, esto se hará restando 32 unidades a la señal fifo1 ya que si antes del desplazamiento valía 43 ahora pasará a valer 11, pero esto se realizará más adelante, solo ha sido un ejemplo para una mejor comprensión. Lo que si se hace en este momento es el desplazamiento de 32 bits hacia la izquierda de la señal buff1, esto se hará cogiendo los bits que se han quedado fuera, del 32 al 43 concatenados con ceros hasta llegar a 32 bits y ese pasará a ser el nuevo valor de buff1, en caso que fifo1 no haya sobrepasado el bit 32, es decir se han enviado todos los bits la señal buff1 pasara a valer cero. Entonces se pasará al reposo.

En caso contrario, se comprobará en qué estado se encuentra la MDE, en caso que se encuentre en Escribir2_1 el proceso Buffer1 comprueba que el FSL de salida esté disponible, es decir FSL3_M_Full = 0, en tal caso se pone a nivel alto la señal env_buff1. Como se ha realizado una transmisión, como se ha dicho antes, habrá que disminuir en 32 el valor de fifo1, que como se recuerda es la señal que indica a partir de que bit se copian los datos en buff1. En caso que el bus de comunicación está lleno el proceso continuará, si el bus FSL está lleno el proceso Buffer1 **no se quedara esperando a que esté disponible, sino que continuará con el bucle**, pero como la MDE sí que espera a que el bus este libre para escribir, el proceso Buffer seguirá con el bucle hasta que se pueda escribir en el FSL por lo que el diseño funcionará correctamente.

En caso que o bien la MDE no esté en el estado Escribir2_1 o bien el bus FSL esté lleno, el proceso Buffer1 pasará a comprobar la siguiente condición. Dicha comprobación será si la MDE está en el estado Volver1, la señal fin1 está a nivel alto (indicando que ya se han procesado todos los datos) y si el valor fifo1 es distinto de cero (indicando que hay algún dato copiado en buff1 listo para ser enviado), en tal caso ya se han descomprimido todos los datos y se dispone a hacer la última transmisión por el FSL, se comprobará si este está disponible y en caso de estarlo (FSL_M_Full = 0), se pondrá a nivel alto la señal env_buff1 y se inicializará la señal fifo1 a cero.

En el supuesto que no se den ninguna de las dos condiciones anteriores el proceso Buffer1 evaluará si la MDE se encuentra en el estado Escribir1_1, en caso positivo indica que se ha procesado un dato y se copiará dicho dato en buff1, en los bits indicados por fifo1, es decir **empezará copiando el valor en el bit fifo1 y terminará en el bit fifo1+12** (ya que son 12 bits).

Por último si no se ha dado ninguna de las condiciones de las descritas hasta ahora, el proceso Buffer1 comprobará si la MDE está en el estado Preset_cont1, en tal caso se actualizará la señal fifo1, como el estado anterior a Preset_cont1 es Escribir1_1, esto quiere decir que se acaba de copiar un dato en buff1, por lo que se sumará 12 al valor de fifo1 para que la próxima copia de dato en buff1 no se realice encima de la anterior. En caso que no se dé esta condición tampoco, el bucle del proceso continuará con la primera condición descrita en este apartado.

En caso de tratarse de los otros dos coprocesadores encargados de realizar la segunda etapa de descompresión, éstos descomprimen datos de 10 bits, entonces lo único que varía es el tamaño del campo, es decir en el FSL de salida de 32 bits se almacenan datos de 10 bits, cuando se incrementa fifo1 para hacer una correcta copia de los datos en vez de incrementarse un valor 12 se incrementa un valor 10, por lo demás los tres coprocesadores tienen un funcionamiento idéntico.

Al igual que en los coprocesadores de la primera etapa de descompresión, en esta segunda etapa se han usado señales internas que luego se han igualado su valor al de las señales de los puertos de comunicación FSL para hacer una correcta transmisión de datos, ya sea de entrada o de salida.

Dichas asignaciones son las siguientes:

- **FSL2_S_Read = '1':** cuando la MDE está en el estado Leer_contador1, es decir cuando está en el estado de leer el contador se pide al FSL que mande el dato para poder leerlo. Este dato es enviado por la primera etapa de descompresión. En cualquier otro estado la señal vale '0'.
- **FSL1_S_Read = '1':** cuando la MDE está en el estado Leer_dato1, cuando se quiere leer un nuevo dato se pide al FSL que mande dicho dato. Este dato es enviado por Microblaze. En cualquier otro estado dicha señal vale '0'.
- **FSL_M_Write = env_buff1**: el valor que indica cuando se han de enviar los datos por el bus FSL de salida se iguala al valor de una señal interna, es decir, cuando en el proceso Buffer1 se pone a nivel alto env_buff1 a su vez se pone a nivel alto FSL_M_Write, por lo que el valor que esté en FSL_M_Data se enviará.
- **FSL_M_Data = buff1**: el dato que se envía a través del FSL de salida es el valor que hay almacenado en los 32 bits menos significativos de la señal interna buff1, que contienen la información de los valores de los datos descomprimidos de 12 bits empaquetados en mensajes de 32 bits.

Con esto queda descrito con detalle el funcionamiento de las máquinas de estado y procesos de los seis coprocesadores necesarios para hacer la descompresión doble.

4.4. DEFINICION DEL INTERFACE SOFTWARE

Para que la comunicación entre el Microblaze y los coprocesadores se pueda establecer correctamente ha sido necesario implementar una interfaz software en lenguaje de programación C. Dicho código se encarga de tres tareas fundamentales, la primera es enviar los datos comprimidos desde Microblaze a los coprocesadores correspondientes, una vez éstos son descomprimidos se encarga de recibirlos de vuelta dichos datos y una vez recibidos se mostrarán por pantalla.

El código C comienza con la declaración de las librerías que se van a utilizar, algunas asociadas al propio código C y otras asociadas al proyecto que se ha implementado que se crean una vez se compila el código VHDL. Estas librerías son las siguientes:

```
#include "Desc_primario_1.h"
#include <stdio.h>
#include <stdlib.h>
#include "xparameters.h"
```

En caso de querer utilizar más de un FSL de entrada o de salida del Microblaze será necesario modificar alguna librería, es el caso de “Desc_primario_1.h” y “xparameters.h”. En el proyecto inicial se creó un tutorial para añadir un FSL de entrada al Microblaze, en este proyecto se ha ampliado para el caso que se quiera utilizar más de un FSL de salida del Microblaze ya que para tener un diseño modular con varios coprocesadores era necesario. Este tutorial se añadirá al final del proyecto como un apéndice por lo que no se entrará en esos detalles ahora.

A continuación se definen unas instrucciones necesarias para la escritura y lectura de datos a través del FSL, estas definiciones se consiguen con las instrucciones **#define WRITE_DESCOMP_0(val)** donde se indicará en qué FSL se desea escribir, y por otra parte **#define READ_DESCOMP_0(val)** donde se indicará que FSL se desea leer. De esta forma como en nuestro proyecto tenemos por cada campo tres FSL de escritura y uno de lectura, contando con todos los campos se obtiene:

```
#define WRITE_DESCOMP_0(val)  write_into_fsl(val, XPAR_FSL_DESCOMP_0_INPUT_SLOT_ID)
#define WRITE_DESCOMP_1(val)  write_into_fsl(val, XPAR_FSL_DESCOMP_1_INPUT_SLOT_ID)
#define WRITE_DESCOMP_2(val)  write_into_fsl(val, XPAR_FSL_DESCOMP_2_INPUT_SLOT_ID)
#define WRITE_DESCOMP_3(val)  write_into_fsl(val, XPAR_FSL_DESCOMP_3_INPUT_SLOT_ID)
#define WRITE_DESCOMP_4(val)  write_into_fsl(val, XPAR_FSL_DESCOMP_4_INPUT_SLOT_ID)
#define WRITE_DESCOMP_5(val)  write_into_fsl(val, XPAR_FSL_DESCOMP_5_INPUT_SLOT_ID)
#define WRITE_DESCOMP_6(val)  write_into_fsl(val, XPAR_FSL_DESCOMP_6_INPUT_SLOT_ID)
#define WRITE_DESCOMP_7(val)  write_into_fsl(val, XPAR_FSL_DESCOMP_7_INPUT_SLOT_ID)
#define WRITE_DESCOMP_8(val)  write_into_fsl(val, XPAR_FSL_DESCOMP_8_INPUT_SLOT_ID)
#define READ_DESCOMP_0(val)  read_from_fsl(val, XPAR_FSL_DESCOMP_0_OUTPUT_SLOT_ID)
#define READ_DESCOMP_1(val)  read_from_fsl(val, XPAR_FSL_DESCOMP_1_OUTPUT_SLOT_ID)
#define READ_DESCOMP_2(val)  read_from_fsl(val, XPAR_FSL_DESCOMP_2_OUTPUT_SLOT_ID)
```

A continuación comienza el main del programa. En primer lugar se declararan las variables que se van a usar y a continuación se inicializarán. En el programa se utilizan muchas variables pero para hacer esta explicación se realizará un ejemplo con un número reducido:

```
main()
{       unsigned int N1, CC1_1, DC1_1, D1_1, D2_1, CC1_2, DC1_2, DC2_2, D1_2,CC1_3, DC1_3 , D1_3,
D2_3;
        unsigned int output_0[8], output_1[8], output_2[8];
        int i, j;

N1 = 0x0000002; //Número de datos que se van a procesar (todos los números son en hexadecimal)
CC1_1 = 0x00000001; //Contador de contador del primer campo de la primera etapa descompresión
DC1_1 = 0x00000001; //Dato de contador del primer campo de la primera etapa descompresión
CC1_2 = 0x00000001; //Contador de contador del segundo campo de la primera etapa descompresión
DC1_2 = 0x00000001; //Dato de contador del segundo campo de la primera etapa descompresión
CC1_3 = 0x00000001; //Contador de contador del tercer campo de la primera etapa descompresión
DC1_3 = 0x00000001; //Dato de contador del tercer campo de la primera etapa descompresión
D1_1 = 0x00000123; //Dato del primer campo de la segunda etapa de descompresión
D2_1 = 0x00000876; //Dato del primer campo de la segunda etapa de descompresión
D1_2 = 0x00000078; //Dato del segundo campo de la segunda etapa de descompresión
D2_2 = 0x00000021; //Dato del segundo campo de la segunda etapa de descompresión
D1_3 = 0x00000078; //Dato del tercer campo de la segunda etapa de descompresión
D2_3 = 0x00000321; //Dato del tercer campo de la segunda etapa de descompresión
```

En el ejemplo se utilizan valores pequeños de contador de contador y dato de contador ya que sino al descomprimir el contador saldría un valor muy grande de contador y el ejemplo sería demasiado largo, para facilitar el ejemplo como tiene que haber un el mismo número de datos descomprimidos en cada campo, se ha puesto los valores de contador de contador y dato de contador iguales en cada campo, aunque en la realidad podrán ser diferentes, por ejemplo el segundo campo se puede repetir dos veces (tres datos consecutivos iguales) y el tercer campo tener tres datos consecutivos distintos.

A continuación una vez declarados los valores que se van a enviar, mediante la instrucción "**WRITE_DESCOMP**" se escribirá en el FSL correspondiente los datos anteriormente declarados. En el ejemplo:

```
WRITE_DESCOMP_0(N1); //Escribe en Desc_primario_1 el valor N1
WRITE_DESCOMP_3(N1); //Escribe en Desc_primario_2 el valor N1
WRITE_DESCOMP_6(N1); //Escribe en Desc_primario_3 el valor N1
WRITE_DESCOMP_1(CC1_1); //Escribe en Desc_primario_1 el valor CC1_1
WRITE_DESCOMP_1(DC1_1); //Escribe en Desc_primario_1 el valor DC1_1
WRITE_DESCOMP_4(CC1_2); //Escribe en Desc_primario_2 el valor CC1_2
WRITE_DESCOMP_4(DC1_3); //Escribe en Desc_primario_2 el valor DC1_2
WRITE_DESCOMP_7(CC1_3); //Escribe en Desc_primario_3 el valor CC1_3
WRITE_DESCOMP_7(DC1_2); //Escribe en Desc_primario_3 el valor DC1_3
WRITE_DESCOMP_2(D1_1); //Escribe en Desc_secundario_1 el valor D1_1
WRITE_DESCOMP_2(D2_1); //Escribe en Desc_secundario_1 el valor D2_1
WRITE_DESCOMP_5(D1_2); //Escribe en Desc_secundario_2 el valor D1_2
WRITE_DESCOMP_5(D2_2); //Escribe en Desc_secundario_2 el valor D21_2
WRITE_DESCOMP_8(D1_3); //Escribe en Desc_secundario_3 el valor D1_3
WRITE_DESCOMP_8(D2_3); //Escribe en Desc_secundario_3 el valor D2_3
```

El siguiente paso es almacenar en las variables declaradas los valores resultantes de la descompresión. Esta operación se realiza mediante la instrucción "**READ_DESCOMP**". Hay que indicar el número de datos que se van a recibir. Si se indica un valor superior al real el código se quedará bloqueado ya que esperara una lectura que nunca se va a realizar porque el descompresor ya ha acabado con el proyecto, en cambio si el valor es inferior, aparentemente todo funcionará correctamente pero los últimos datos no se habrán copiado en las variables correspondientes.

Por último sólo queda mostrar por pantalla los datos recibidos, esto se hará mediante la instrucción "**xil_printf**" cuyo funcionamiento es el mismo que el "printf" del código C. Por lo tanto en el ejemplo el código restante será el siguiente:

```
for (i=0; i<1; i++)
{
	READ_DESCOMP_0(output_0[i]); // Lectura del dato de salida de Desc_secundario_1
	xil_printf("Salida modulo_1 %d: %X \r\n",i+1,output_0[i]);
}
xil_printf("\r\n");
for (i=0; i<1; i++)
{
	READ_DESCOMP_1(output_1[i]); // Lectura del dato de salida de Desc_secundario_2
	xil_printf("Salida modulo_2 %d: %X \r\n",i+1,output_1[i]);
}
xil_printf("\r\n");
for (i=0; i<1; i++)
{
	READ_DESCOMP_2(output_2[i]); // Lectura del dato de salida de Desc_secundario_3
	xil_printf("Salida modulo_3 %d: %X \r\n",i+1,output_2[i]);
}
```

En caso de ser tramas de datos más largas, como el primer campo es de 12 bits y los otros dos de 10, el primer campo necesitará menos datos para llenar un mensaje de 32 bits, por lo tanto será necesario leer más veces los datos de Desc_secundario_1 que de los otros dos coprocesadores de la segunda etapa.

Por último para tramas largas de datos se pide que muestre que número de proceso acaba de completar y el número total de datos procesados:

```
xil_printf("Proceso %d completado\r\n\r\n",j+1);
xil_printf("Datos procesados: %d\r\n\r\n",d*(j+1));
```

Por lo tanto el ejemplo anterior tendría el siguiente resultado:

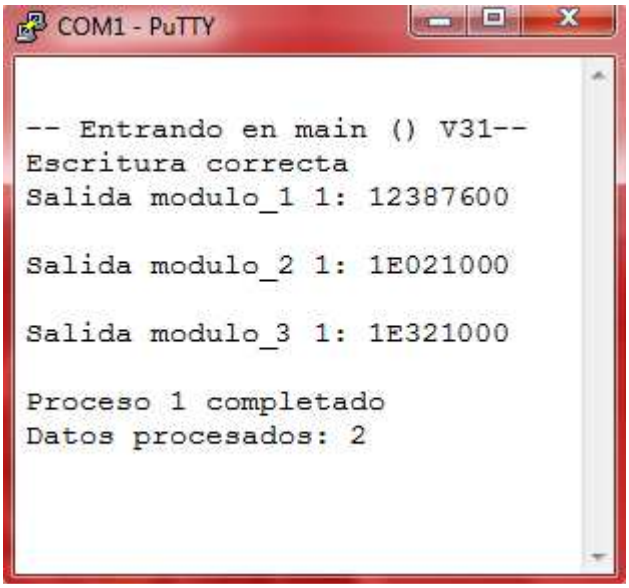


Figura 20: Ejemplo descompresión 2 datos.

- Salida modulo_1: corresponde a la salida del segundo coprocesador del primer campo (Desc_secundario_1) los datos son de 12 bits.
- Salida modulo_2: corresponde a la salida del segundo coprocesador del segundo campo (Desc_secundario_2) los datos son de 10 bits.
- Salida modulo_3: corresponde a la salida del segundo coprocesador del tercer campo (Desc_secundario_3) los datos son de 10 bits.

Como se observa, en este caso sólo se han descomprimido dos datos, en el primer caso se ve claro que están los 12 bits del primer dato y a continuación los 12 bits del segundo dato. En el segundo y tercer dato aparentemente está mal pero da error a confusión al estar el resultado en hexadecimal ya que el dato son de 10 bits, si se comprueba bit a bit el resultado es correcto, esto se da por lo siguiente: el dato 78 en decimal es “00 0111 1000” pero como el buffer de salida agrupa los datos, ese número se copia en la parte izquierda del buffer y pasa a ser “0001 1110 00” y a continuación pasa a escribirse el siguiente por lo tanto en el primer caso tenemos el número “0111 1000” = 78 y en el segundo “0001 1110” = 1E.

4.5. INTEGRACIÓN DE LA ARQUTECTURA COMPLETA

Este proyecto tiene dos objetivos principales, el primero es el implementar un algoritmo de descompresión en una FPGA mediante lenguaje de descripción Hardware (VHDL) que ya se ha explicado a distintos nivel de detalle y como se ha visto en el último apartado se ha comprobado que funciona correctamente. Dicha implementación resulta como continuación de un primer proyecto fin de carrera que consistió en implementar de igual manera un algoritmo de descompresión.

El segundo objetivo del proyecto es, una vez los dos diseños funcionan correctamente, hacer un único diseño que contenga los dos algoritmos, el de compresión y descompresión.

4.5.1. Descripción de la arquitectura en alto nivel

Como en este proyecto ya se ha descrito el algoritmo con un alto nivel de detalle y en el proyecto de compresión de datos también se hizo, únicamente se va a describir a algo nivel ya que el funcionamiento es el mismo, la única diferencia es que un único diseño incluye los dos algoritmos.

En principio **existen dos opciones**, la primera conectar los dos diseños en serie, es decir uno detrás de otro, o bien, la segunda opción es conectar los dos diseños en paralelo. La primera opción puede ser interesante para comprobar que tanto el algoritmo como los diseños implementados funcionan correctamente pero tener un compresor conectado directamente a un descompresor tiene poca utilidad.

La segunda opción representa un a mayor elaboración a la hora de comprobar el algoritmo y el diseño pero en cambio **tiene un uso práctico** que es tener en una FPGA ambos diseños totalmente independientes, es decir se puede utilizar tanto como compresor o como descompresor. Tiene mucha más utilidad ya que tienes una FPGA que se puede cambiar de ubicación con facilidad y según las necesidades puede realizar una función u otra, esto puede resultar útil ya que una FPGA no es un aparato fijo, por sus pequeñas dimensiones es fácil de cambiar de ubicación. Por estas razones se ha decidido implementar ambos diseños en paralelo y de forma totalmente independiente.

A la hora de implementar ese diseño ha surgido un problema. Dicho problema consiste en que el **número máximo de FSL** que se pueden poner en un diseño al Microblaze es de 18, 9 maestros y 9 esclavos, teniendo en cuenta que el descompresor usa 9 FSL maestro y dos FSL esclavo para el Microblaze y el compresor utiliza 1 FSL maestro y 9 FSL esclavo para el Microblaze. A parte los diseños utilizan FSL para conectar los distintos coprocesadores pero para esos no existe problema alguno. Para solventar el problema se ha decidido reducir el número de campos, se ha eliminado un campo por lo que ahora solo habrá dos, en principio se ha elegido un tamaño de campo de 16 bits cada uno pero esto es totalmente configurable por lo tanto por una parte habrá un descompresor de dos campos y otro compresor de dos campos, el número total de FSL conectados al Microblaze es de 7 FSL maestro y 8 FSL esclavo por lo que el diseño ya es correcto.

Por consiguiente el esquema general de la arquitectura de compresión y descompresión de dos campos es el siguiente:

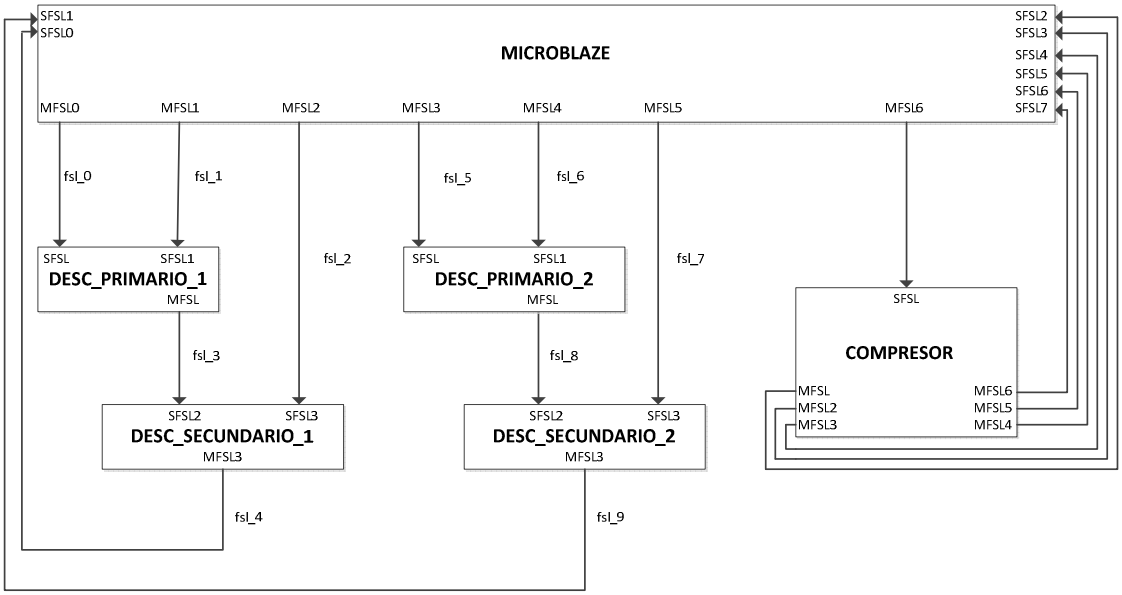


Figura 21: Esquema arquitectura de compresión y descompresión doble.

El esquema de la compresión y descompresión doble consta de seis módulos, el primero se trata del Microblaze, los dos siguientes son los debidos a la primera descompresión y los dos siguientes los correspondientes a la segunda descompresión y por último el restante es el debido a la doble compresión.

Cada módulo de la primera descompresión está conectado mediante dos FSL al Microblaze. La salida de los módulos de la primera descompresión se envía a la segunda descompresión, que se trata del valor de "Contador" necesario en estos módulos. En este punto necesita también que el Microblaze mande el valor de "Dato" para reconstruir el dato inicial (antes de la compresión). Por otra parte está el módulo de la compresión doble que recibe los datos por un único FSL, los procesa y saca el valor de la compresión por 6 FSL (3 por cada campo) dichos datos corresponden al valor de "Dato", al valor de "Contador" y al valor de la compresión del contador, es decir "Contador de contador". Por lo tanto esta arquitectura consta de 17 FSL de comunicación.

El funcionamiento de este diseño para comprobar el correcto funcionamiento tanto del algoritmo como de los diseños (compresión y descompresión) consta de **dos fases**. En la primera fase Microblaze manda un conjunto de datos a Compresor, este coprocesador se encarga de hacer la compresión doble de dichos datos, una vez que los ha procesado manda de vuelta los resultados al Microblaze. En segundo lugar Microblaze manda esos datos recibidos a la parte de la descompresión doble, es decir, mandará el valor de "Contador de contador" a Desc_primario_1 y Desc_primario_2, que descomprimirán en valor de "Contador" que será enviado según corresponda a Desc_secundario_1 y Desc_secundario_2 junto con el valor de "Dato" enviado por Microblaze. Una vez que estos dos últimos coprocesadores mandan sus resultados a Microblaze, se puede comprobar si la compresión y posterior descompresión han sido satisfactorios ya que estos últimos datos recibidos deben ser iguales que los primeros datos enviados por Microblaze.

El funcionamiento interno de cada coprocesador es exactamente igual que el descrito en sus correspondientes capítulos de la memoria por eso no se entra más en detalle.

4.5.2. Descripción interface Software

Al igual que en el caso del diseño de descompresión doble, para que la comunicación entre el Microblaze y los coprocesadores se pueda establecer correctamente ha sido necesario implementar una interfaz software en lenguaje de programación C. Dicho código se encarga de tres tareas fundamentales, la primera es enviar los datos comprimidos desde Microblaze a los coprocesadores correspondientes, una vez éstos son comprimidos o descomprimidos se encarga de recibirlos de vuelta dichos datos y una vez recibidos se mostrarán por pantalla.

En este caso el programa en C tiene la misma estructura que lo que se ha explicado en anteriores capítulos, es decir las librerías son las mismas, la manera en que se escribe o se lee de los FSL es la misma por lo que no se va a entrar en detalle de ver cómo es el código C, sino que se va a realizar una visión más global de como se ha implementado el código C para validar la integración de ambos proyectos.

El diseño elegido de la interconexión en paralelo entre compresor y descompresor hace que el interface software sea más complicado, ya que si se hubiese conectado en serie sólo habría que escribir datos en el coprocesador correspondiente a la compresión y leer los datos de salida de los coprocesadores correspondientes a la descompresión. Pero como se ha decidido un diseño más versátil para que pueda tener una función real el programa **se divide en dos partes claramente identificables**. Al ser dos diseños totalmente independientes el programa engloba los programas que se utilizaban para estudiar tanto el compresor como el descompresor individualmente, la única diferencia es que los datos obtenidos en el compresor son los que se utilizan para el descompresor.

En tal caso el programa sigue los siguientes pasos. En primer lugar se declaran las variables que se van a enviar al procesador, son variables de 32 bits que posteriormente el compresor dividirá en dos variables de 16 bits (el tamaño de los campos). A continuación dichos datos serán enviados al coprocesador encargado de hacer la compresión. Una vez se haya realizado dicha compresión, Microblaze copiará los valores de salida del compresor. Cuando el proceso de compresión haya terminado Microblaze comenzará a enviar a Desc_primario_1, Desc_secundario_1, Desc_primario_2 y Desc_secundario_2 los datos obtenidos en la compresión. Una vez se haya terminado la descompresión, los dos últimos coprocesadores enviarán de vuelta los datos obtenidos. Estos datos se imprimirán por pantalla para comprobar que son los mismos que se enviaron a la etapa de compresión.

A continuación se va a mostrar un ejemplo de una trama de 16 datos que se repiten por parejas, es decir, dos datos consecutivos iguales, y los siguientes dos datos son iguales entre sí pero distintos a los dos primeros. En ese caso en la etapa de compresión se obtendrían 8 datos distintos y 8 contadores iguales, al comprimir los contadores se obtendría un único valor de dato de contador y un contador de contador que tendría el valor de 7 (7 repeticiones es decir 8 contadores iguales). Una vez realizada la compresión dichos valores se transmiten a la etapa de descompresión y se obtienen los 16 datos iniciales.

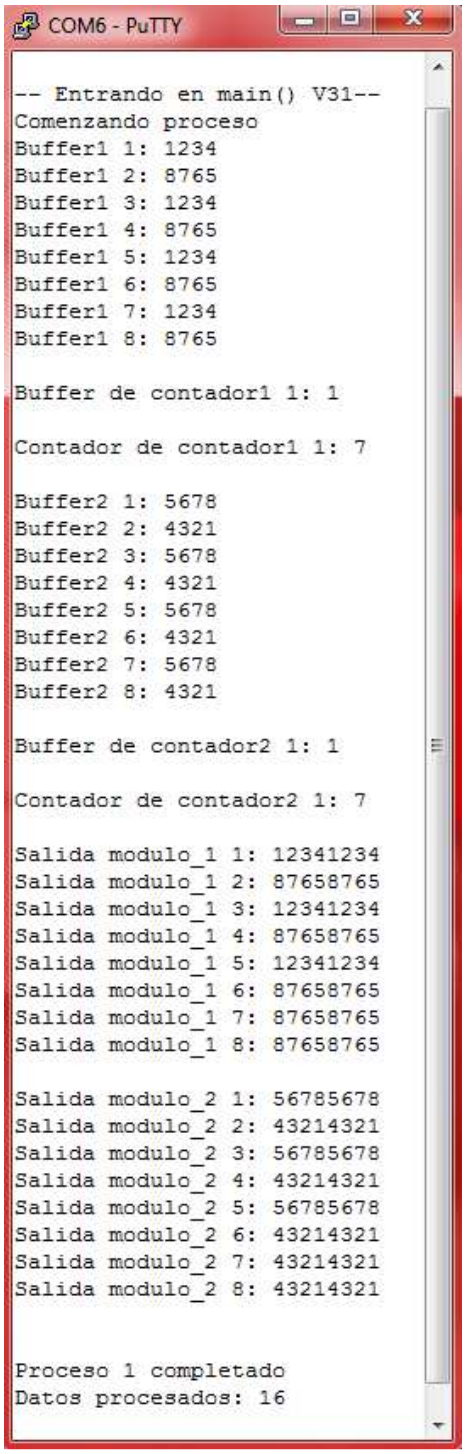


Figura 22: Ejemplo funcionamiento diseño completo.

Como se puede observar la salida de la etapa de descompresión sólo tiene 8 valores pero como los datos son de 16 bits en cada transmisión se envían 2 datos. Como se puede observar en la Figura 22 los datos de salida son los mismos que los de entrada por lo que se comprueba que tanto el algoritmo como los diseños de compresión y descompresión funcionan correctamente.

5. ESTUDIO DEL RENDIMIENTO

En este capítulo se va a explicar los pasos seguidos antes de probar el diseño en la FPGA y posteriormente se va a hacer una descripción de las pruebas realizadas y sus resultados. Estas pruebas incluyen un análisis de la velocidad de la arquitectura de descompresión en varios dispositivos, un análisis del consumo de energía en la arquitectura de descompresión simple y por último ambos dos análisis en el diseño completo. También se hará un análisis del coste hardware de los diseños en diferentes dispositivos. Para terminar se hará una comparación entre la arquitectura hardware descrita en VHDL, una arquitectura que se ha implementado en código C en el Microblaze y por último una arquitectura software ya existente.

5.1. VALIDACIÓN DE LA ARQUITECTURA

En este apartado se va a describir la herramienta utilizada en primer lugar para diseñar el código, que es anterior a la prueba directa del diseño en la propia FPGA.

Se empezó a hacer el diseño directamente en la FPGA, pero debido al tiempo que tarda en compilar el programa y que en caso de que el diseño no funcionara no se podía averiguar cuál era el fallo se decidió utilizar la herramienta **Quartus II**, ya que había conocimientos previos adquiridos en otra asignatura y la posesión de una versión gratuita del programa.

Este programa permite implementar un diseño en VHDL, compilarlo rápidamente. Además tiene un simulador muy visual en el que se puede comprobar el correcto funcionamiento del diseño y en caso contrario permite averiguar en poco tiempo donde está el problema.

5.1.1. Quartus II

Quartus II es una herramienta de software desarrollada por Altera para el diseño, análisis y la síntesis de HDL. Permite al desarrollador compilar diseños, realizar análisis temporales, examinar diagramas RTL y configurar el dispositivo de destino con el programador.

El empleo de Quartus II permite a los diseñadores utilizar los dispositivos HardCopy Stratix integrados en el programa para realizar pruebas sobre el código diseñado. Por lo tanto es una manera de verificar el rendimiento que tendría en una FPGA.

Aunque el entorno de desarrollo usado ha sido el XPS, propiedad de Xilinx se ha utilizado la herramienta Quartus II ya que dicha herramienta tiene un tiempo de compilación mucho menor que el XPS y debido a que tiene un entorno gráfico de simulación fácil de usar con el cual se había utilizado con anterioridad y su **manejo era conocido**. Su uso también ha sido debido a la posesión de una versión gratuita por lo que no era necesario adquirir licencia. En caso contrario se podía haber utilizado bancos de prueba con las herramientas desarrolladas por Xilinx.

A diferencia del XPS, Quartus II realiza una rápida compilación y depuración de errores, mientras que XPS tarda alrededor de 10 minutos en compilar el código VHDL. Utilizando Quartus II el proceso se acorta considerablemente, llegando a durar apenas 30 segundos.

Además, posee una herramienta de simulación que permite ver el funcionamiento del código mediante diagramas de tiempo, donde se puede comprobar en qué momento se activan o desactivan las señales o ver donde se encuentra una máquina de estados. Xilinx también dispone de una herramienta de simulación, pero es necesario un banco de pruebas, con el que no existía ninguna familiarización, mientras en Quartus II no es necesario, puesto que también permite la activación y desactivación de las señales de entrada manualmente. Es una herramienta cuyo uso es rápido e intuitivo.

Por estas razones el código VHDL en primer lugar fue desarrollado en Quartus II ya que al principio es normal que haya numerosos errores de compilación o diseños erróneos, una vez que se comprobó que el diseño era válido en el simulador de Quartus II se pasó dicha implementación al XPS y a la FPGA utilizada. Con esto se consiguió una mayor agilidad a la hora de avanzar en el proyecto.

A la hora de implementar el diseño se ha seguido una estrategia **modular ascendente**, es decir, se ha empezado de un diseño básico y se han ido añadiendo módulos independientes hasta tener el diseño completo. En primer caso se diseñó un único coprocesador con la función de realizar una descompresión simple de un solo campo. Una vez éste funcionó correctamente, se prosiguió con un diseño de dos coprocesadores encargados de hacer una descompresión doble con un solo campo. Cuando dicha implementación fue funcional se pasó al diseño definitivo que consistía en repetir el diseño existente con dos nuevos tamaños de campo. Una vez se realizó dicha implementación se verificó con la herramienta Quartus II y una vez los errores fueron depurados se pasó a hacer pruebas con la FPGA. Cada vez que se hizo un diseño o se modificó algo siempre la manera de actuar fue primero verificar con Quartus II que los cambios o ampliaciones realizadas funcionaban correctamente y en segundo lugar hacer una prueba definitiva en la FPGA.

A continuación se va a mostrar un ejemplo de una simulación en Quartus II, para comprobar que es fácil e intuitivo de usar:

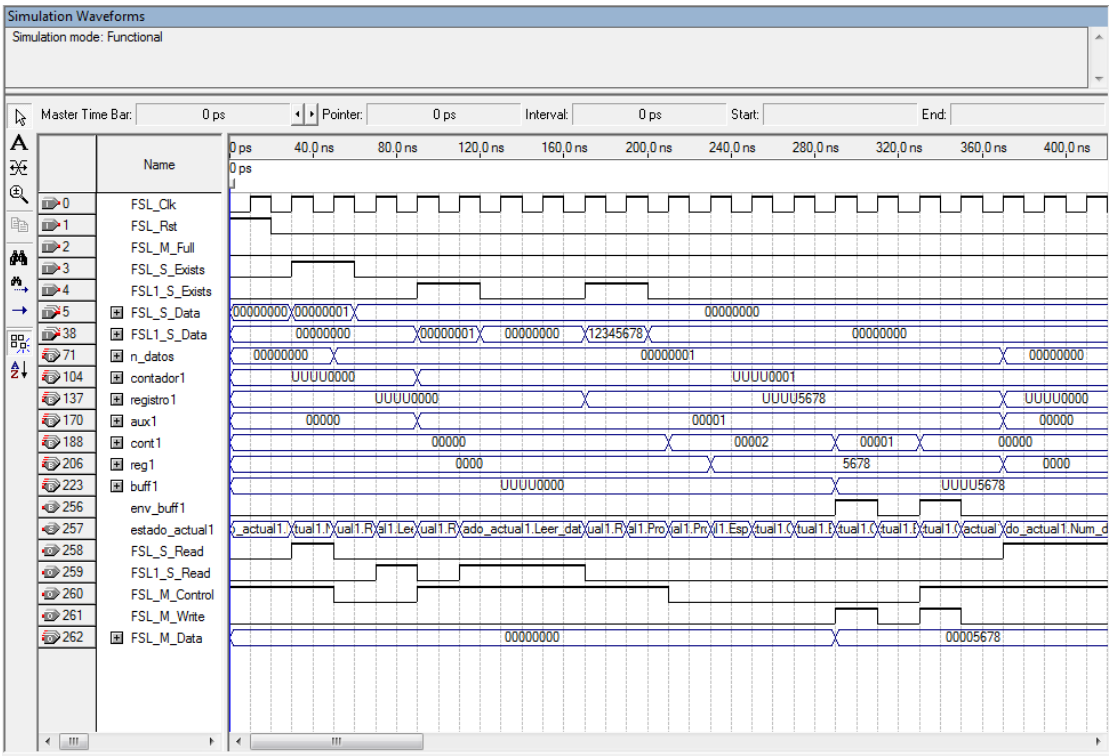


Figura 23: Ejemplo simulación Quartus II.

Este ejemplo muestra una descompresión simple de un único campo de 16 bits. En primer lugar hay que definir la señal FSL_Clk que será el reloj de la FPGA. A continuación se hace un reset del diseño para asegurar que la MDE comienza en el estado adecuado. Las señales FSL_S_Exists y FSL1_S_exists se configuran como el usuario desee, cada vez que se quiere que la MDE lea un dato habrá que poner la señal correspondiente a nivel alto. Después hay que configurar la señal FSL_S_Data que contiene el valor del número de datos que se van a procesar y la señal FSL1_S_Data que la primera vez que se lea contiene el valor del contador y la segunda vez el valor de dato, en este ejemplo se ha configurado de manera que sólo se va a procesar 1 dato, el contador es 1 (el dato se repite una vez) y el dato es 5678 (16 bits menos significativos).

Como se observa en la Figura 23 cuando se lee del primer FSL dicho valor se copia en la señal n_datos (número de datos que se van a procesar), la siguiente lectura corresponde al segundo FSL que se copia en la señal contador1, y en la última lectura el valor del segundo FSL se copia en la señal registro1. Cuando se realiza la primera lectura de contador la señal aux1 aumenta en una unidad su valor, indicando que **se está descomprimiendo el primer dato**.

A continuación los 16 bits menos significativos de la señal contador1 se copian en cont1, se concatena un cero por la izquierda y se le suma la unidad a dicho valor, por lo que ahora el valor de cont1 es el número de veces que hay que escribir el dato para hacer una correcta descompresión. De la misma manera los 16 bits menos significativos de la señal registro1 se copian en la señal reg1, **éste es el tamaño del campo**, en caso de tener un tamaño distinto, por ejemplo 12 bits, se copiarían los 12 bits menos significativos.

Por último se escribe en el FSL de salida el valor de reg1, cada vez que se realiza una escritura se disminuye en una unidad el valor de cont1 hasta que dicho valor llega a cero. Como sólo se procesa un dato aux1 y n_datos tienen el mismo valor por lo que el proceso ha terminado y en la última escritura se activa la señal FSL_M_Control que indicará al siguiente coprocesador que ese es el último dato a descomprimir.

Como ya se ha dicho anteriormente una vez realizadas estas pruebas se pasó a implementar el diseño en distintas FPGAs. Antes de realizar pruebas de velocidad y de consumo energético se validó el algoritmo y los diseños realizados. Para ello se mandó una trama de datos conocida al compresor, se ejecutó la etapa de compresión y con los datos obtenidos se ejecutó la etapa de descompresión, si el proceso funcionó correctamente los datos obtenidos deberían ser iguales que los mandados al compresor, como se puede comprobar en la Figura 22, esto es lo que sucedió, por lo que tanto el algoritmo de compresión como las dos arquitecturas implementadas quedan totalmente validadas y se puede afirmar que se realiza una compresión–descompresión, sin pérdidas.

5.2. ANÁLISIS DE VELOCIDAD DE LA ARQUITECTURA DE DESCOMPRESIÓN EN DISTINTAS FPGAs

Para realizar el análisis de la velocidad de la arquitectura se ha diseñado el siguiente procedimiento. En primer lugar se ha hecho una prueba de descompresión de 16 datos conocidos para verificar que dicha descompresión ha sido satisfactoria. Una vez hecha esa comprobación se ha hecho un bucle en el programa C de Microblaze para que repita esa operación de descompresión un gran número de veces, ya que una única vez no da tiempo a medirlo. Ese número de repeticiones se refiere a una magnitud de decenas o centenas de millones de repeticiones. El método de medir el tiempo se ha realizado mediante un cronómetro que aparentemente resulta un aparato sin demasiada precisión pero como las pruebas que se han realizado están en torno a los diez minutos, un error de un segundo sobre seiscientos es despreciable.

Por otra parte existen varias formas de realizar la comunicación FSL. De forma síncrona con los comandos “putfsl()” y “getfsl()”, o de forma asíncrona con los comandos “nputfsl()” y “ngetfsl()”.

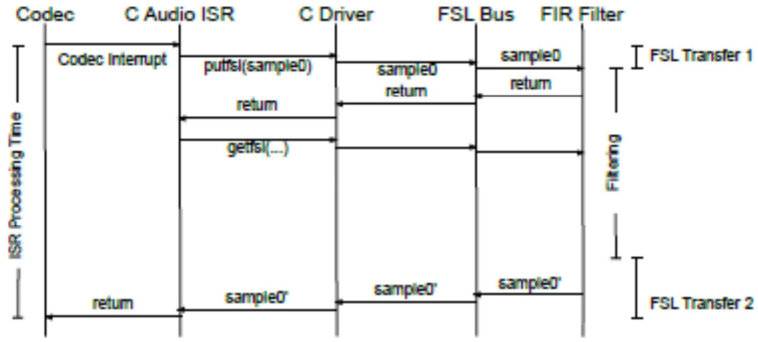


Figure 6. Blocking Filtering Process Sequence Diagram

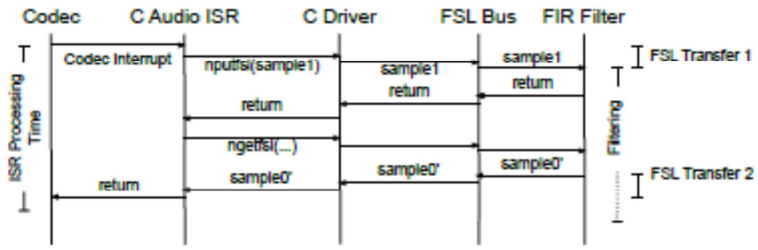


Figure 7. Non-Blocking Filtering Process Sequence Diagram

Figura 24: Diferencia entre comunicación síncrona o asíncrona.

La Figura 24 muestra la diferencia de tiempo que hay que esperar entre una comunicación y la siguiente en un bus FSL dependiendo de si se usa comunicación síncrona o asíncrona (se observa que en el segundo caso el “Filtering” es mucho menor) por lo que se acelera sustancialmente mediante la sustitución de los estándares de bloqueo de transferencia de rutinas síncronas (o de bloqueo) por rutinas asíncronas (o sin bloqueo). La rutina síncrona tiene que esperar hasta que el proceso de filtrado ha terminado, mientras que en el otro caso la rutina vuelve inmediatamente [TH–Filtering].

Por lo tanto en las pruebas se hará una distinción para ver la diferencia entre la escritura síncrona y asíncrona. También se van a mostrar los resultados obtenidos en la implementación en distintas FPGAs.

5.2.1. FPGA Nexys 2

Como ya se ha dicho con anterioridad se ha configurado el programa C de Microblaze para que haga un bucle de millones de repeticiones de una secuencia corta conocida. En primer lugar se imprimen por pantalla los resultados para comprobar que el funcionamiento es correcto y una vez hecho esto se realiza la prueba sin imprimir por pantalla ya que esto ralentiza el proceso. Se han medido los datos enviados por segundo pero interesa más representar los megabytes por segundo que es capaz de descomprimir. Como cada dato es de 32 bits por cada dato por segundo descomprimido se han descomprimido 4 bytes por segundo. Con esto se puede calcular la velocidad en megabytes por segundo.

También se ha comprobado que el tiempo de procesado es lineal, es decir, que proporcionalmente tarda diez veces más en procesar cien millones de datos que diez millones de datos, por lo tanto la velocidad permanece constante y es independiente del número de datos a descomprimir. Por lo que se han hecho pruebas largas para minimizar el error del cronómetro.

Por otra parte se han realizado tres pruebas distintas, por una parte todos los datos a descomprimir son distintos, es decir todos los contadores valen cero, un segundo caso que se trata que los datos sólo se repiten una vez, es decir, los contadores valen 1, y en último caso todos los datos son iguales, es decir un único dato que se repite 15 veces (ya que se envían secuencias de 16 datos). Estas pruebas se han realizado en la FPGA Nexys 2.

En este análisis también se verificará si hay alguna diferencia entre los dos diseños implementados. El primero usa un único FSL para mandar los contadores de contador y los datos de contador y el segundo aprovecha el FSL por el que se manda el número de datos a descomprimir para enviar los contadores de contador y por el otro FSL los datos de contador.

16 datos repetidos 100 millones de veces - 5.96 GB (50 MHz)				
Tipo de datos	**Diseño 1**		**Diseño 2**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	404	15,11	405	15,07
Con una repetición	275	22,19	274	22,28
Iguales	137,6	44,36	137,8	44,29

Tabla 9: Comparativa velocidad del descompresor entre ambos diseños en Nexys 2.

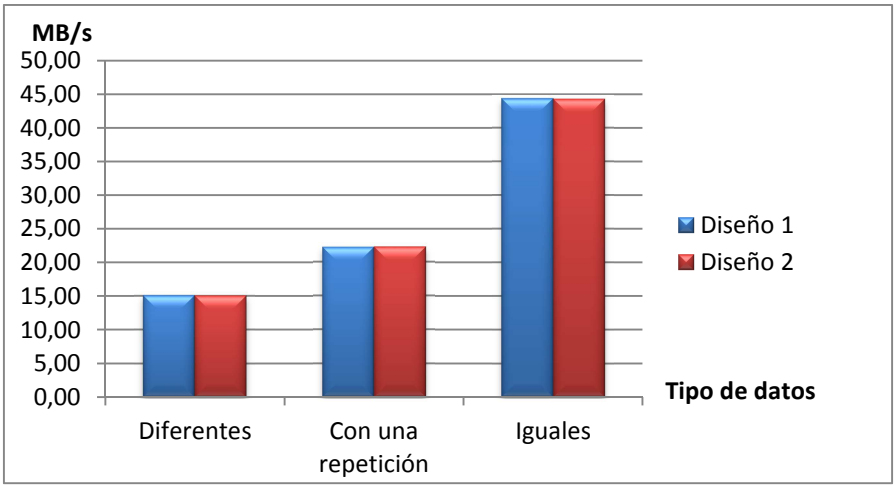


Figura 25: Comparativa velocidad del descompresor entre diseños ambos en Nexys 2.

Como se observa en la Tabla 9 y la Figura 25 el anterior análisis no existe una clara diferencia entre los dos diseños, por lo que se puede afirmar que ambos diseños son igual de eficientes. Todas las pruebas que se han realizado se han hecho con ambos diseños pero al ver que los resultados son exactamente los mismos a partir de ahora consideraremos un único diseño ya que son igualmente eficientes.

La frecuencia de reloj del código VHDL viene determinada por la FPGA que se utiliza, en este caso son 50 MHz, pero la frecuencia de Microblaze se puede configurar. Por otro lado también se puede configurar Microblaze para que, dentro de la misma frecuencia, nos permita tener un diseño que exprima al máximo la frecuencia a consta de que el diseño requiera más recursos de la FPGA. Por lo tanto se ha realizado otra prueba en la que en un mismo diseño se han realizado pruebas de velocidad variando la frecuencia de Microblaze. En primer caso se utilizará la configuración por defecto, 50 MHz, en segundo lugar la configuración de 50 MHz pero a máxima frecuencia, y un tercer caso a 75 MHz a máxima frecuencia. Los resultados son los siguientes:

16 datos repetidos 100 millones de veces - 5.96 GB						
Tipo de datos	50 MHz		50MHz máxima frecuencia		75 MHz máxima frecuencia	
	Tiempo (seg)	Rend (MB/s)	Tiempo (seg)	Rend (MB/s)	Tiempo (seg)	Rend (MB/s)
Diferentes	404	15,11	378	16,15	252	24,22
Con una repetición	275	22,19	260	23,48	173,3	35,22
Iguales	137,6	44,36	137	44,55	91,4	66,78

Tabla 10: Comparativa velocidad del descompresor con distintas frecuencias de Microblaze en Nexys 2.

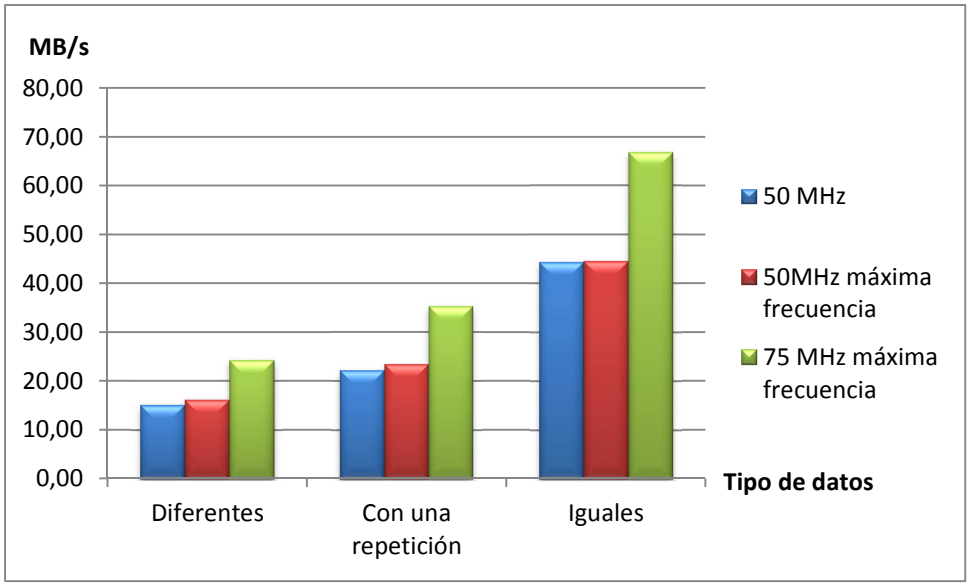


Figura 26: Comparativa velocidad del descompresor con distintas frecuencias de Microblaze en Nexys 2.

Los resultados obtenidos en la Tabla 10 y Figura 26 son interesantes, ya que a mayor frecuencia de escritura y lectura de los datos el diseño tiene una velocidad normal, algo que en principio parece lógico, pero hay que tener en cuenta que en todos los casos la frecuencia de reloj del código VHDL permanece constante. Esto quiere decir que lo que limita el diseño es la comunicación entre Microblaze y los coprocesadores. Teniendo esto en cuenta los datos son los esperados ya que al aumentar un 50% la frecuencia de Microblaze la velocidad también aumenta un 50% aproximadamente.

Otra prueba que se ha realizado es medir la velocidad con distintos tipos de comunicación, síncrona y asíncrona (como se ha explicado en el capítulo anterior). Antes de realizar dicha prueba se ha realizado la comprobación que el diseño funcione en ambos tipos de comunicación ya que en la asíncrona se podría dar el caso de que se perdieran algunos datos ya que no se espera una confirmación que el dato se ha recibido correctamente.

Tras hacer esa comprobación se ha visto que el diseño funciona correctamente en ambos casos por lo que se ha procedido a hacer las pruebas de velocidad con los siguientes resultados:

16 datos repetidos 100 millones de veces - 5.96 GB				
Tipo de datos	**50 MHz síncrona**		**50MHz máxima frecuencia síncrona**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	404	15,11	378	16,15
Con una repetición	275	22,19	260	23,48
Iguales	137,6	44,36	137	44,55
Tipo de datos	**75 MHz máxima frecuencia síncrona**		**50 MHz asíncrona**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	252	24,22	228	26,77
Con una repetición	173,3	35,22	211	28,93
Iguales	91,4	66,78	112	54,50
Tipo de datos	**50 MHz máxima frecuencia asíncrona**		**75 MHz máxima frecuencia asíncrona**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	120	50,86	80	76,29
Con una repetición	139	43,91	92,66	65,87
Iguales	86,2	70,81	57,4	106,33

Tabla 11: Comparativa velocidad del descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Nexys 2.

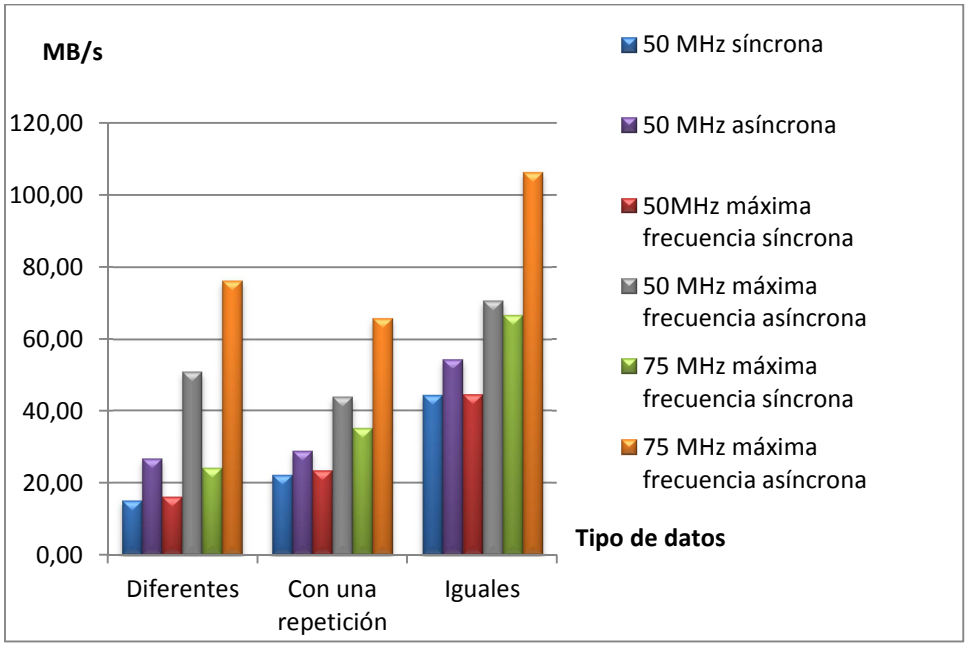


Figura 27: Comparativa velocidad del descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Nexys 2.

De esta prueba de la Tabla 11 y Figura 27 se pueden sacar varias conclusiones. En primer lugar, es extraño ya que la descompresión siendo todos los datos diferentes en caso de comunicación asíncrona, es más rápida que los datos con una repetición, esto se da ya que al cambiar el modo de comunicación, la asíncrona tarda menos en transmitir los datos, y en este caso con datos diferentes se transmiten el doble de datos que si tienen una repetición (hay el doble de contadores), por lo que el diseño VHDL es un poco más rápido con datos con una repetición, pero la comunicación es mucho más rápida ahora que se hace de modo asíncrono. Esto no pasa cuando todos los datos son iguales ya que aunque se envían muchos menos datos el incremento de velocidad del código VHDL en este caso es muy superior al incremento de velocidad en la comunicación por esto el caso de que todos los datos sean iguales es el más favorable.

Por otra parte en el modelo de 50 MHz que viene por defecto la diferencia entre comunicación síncrona o asíncrona es menor que en el resto de los casos, por lo que si se quiere una mejora de velocidad importante hay que configurar el Microblaze en modo de máxima frecuencia. Esto es lo que ocurre en los dos siguientes casos, aunque la velocidad de comunicación es la misma se puede conseguir una mejora en torno del 60% al 215%, dependiendo del tipo de datos que se descompriman.

En resumen, con todas estas pruebas se obtienen varias conclusiones, en primer caso que dependiendo del tipo de datos que se descompriman la velocidad del diseño es distinta. En segundo caso que modificando la frecuencia a la que trabaja Microblaze se consigue aún una velocidad mayor. Y en tercer y último lugar, se puede utilizar una comunicación asíncrona ya que la comunicación también resulta satisfactoria y en algunos casos se puede conseguir más de velocidad. Es interesante estudiar estos casos ya que si lo que importa es la velocidad del algoritmo, en el mejor de los casos puede ser incrementada hasta un 215% más subiendo la frecuencia de Microblaze y utilizando comunicación asíncrona.

5.2.2. FPGA Spartan 3

A continuación se han realizado las mismas pruebas en otro dispositivo, este caso se trata de una FPGA Spartan 3. La placa Nexys 2 y Spartan 3 constan de una FPGA que se trata de una Spartan 3, son distintos modelos pero de la misma familia y el reloj del código VHDL es el mismo (50 MHz).

Se ha comprobado que el diseño también funciona correctamente en esta placa. Al tratarse de FPGAs muy parecidas con la misma frecuencia es de esperar que los resultados sean similares.

16 datos repetidos 100 millones de veces - 5.96 GB (50 MHz)				
Tipo de datos	**Nexys 2**		**Spartan 3**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	404	15,11	405	15,07
Con una repetición	276	22,11	275	22,19
Iguales	137,6	44,36	138,9	43,94

Tabla 12: Comparativa velocidad del descompresor entre distintas FPGAs.

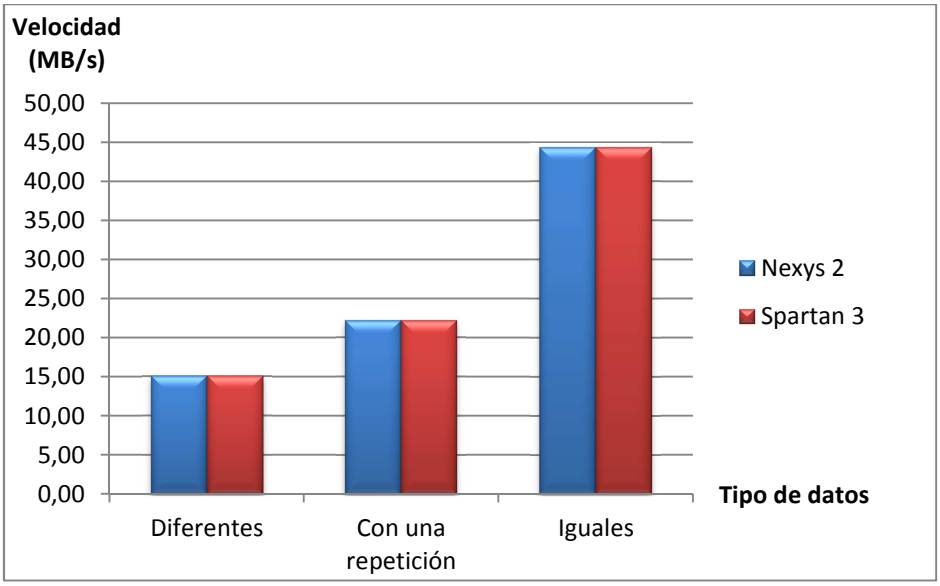


Figura 28: Comparativa velocidad del descompresor entre distintas FPGAs.

Como es de esperar el resultado de la Tabla 12 y la Figura 28 obtenido en ambas placas es exactamente el mismo. Pero aun así es interesante seguir haciendo pruebas ya que la Spartan 3 permite un mayor rango de frecuencias que se pueden asignar a Microblaze, y también más adelante se hará un estudio del consumo energético de distintas placas y con un mismo diseño con misma frecuencia de Microblaze se podrá saber que placa es la que optimiza mejor sus recursos.

Esta FPGA permite un valor máximo de frecuencia de Microblaze de 90 MHz, se han realizado pruebas con los mismos valores que la Nexys dos pero como ya se ha visto, los resultados son los mismos, por lo que sólo se va a mostrar la configuración base (50 MHz), la configuración en la que la Nexys 2 ofrece una mayor velocidad (75 MHz máxima frecuencia), y

la máxima velocidad de la Spartan 3 (90 MHz máxima frecuencia). Se han hecho pruebas tanto en comunicación síncrona como asíncrona, ya que se ha comprobado que se puede obtener un gran aumento en la velocidad de descompresión.

Por lo tanto los datos obtenidos son los siguientes:

16 datos repetidos 100 millones de veces - 5.96 GB				
Tipo de datos	**50 MHz síncrona**		**75MHz máxima frecuencia síncrona**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	404	15,11	252	24,22
Con una repetición	275	22,19	173,3	35,22
Iguales	137,6	44,36	91,4	66,78
Tipo de datos	**90 MHz máxima frecuencia síncrona**		**50 MHz asíncrona**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	210	29,06	228	26,77
Con una repetición	144,3	42,30	211	28,93
Iguales	76,2	80,10	112	54,50
Tipo de datos	**75 MHz máxima frecuencia asíncrona**		**90 MHz máxima frecuencia asíncrona**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	80	76,29	66,6	91,64
Con una repetición	92,67	65,86	77,17	79,09
Iguales	57,4	106,33	47,9	127,42

Tabla 13: Comparativa velocidad del descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Spartan 3.

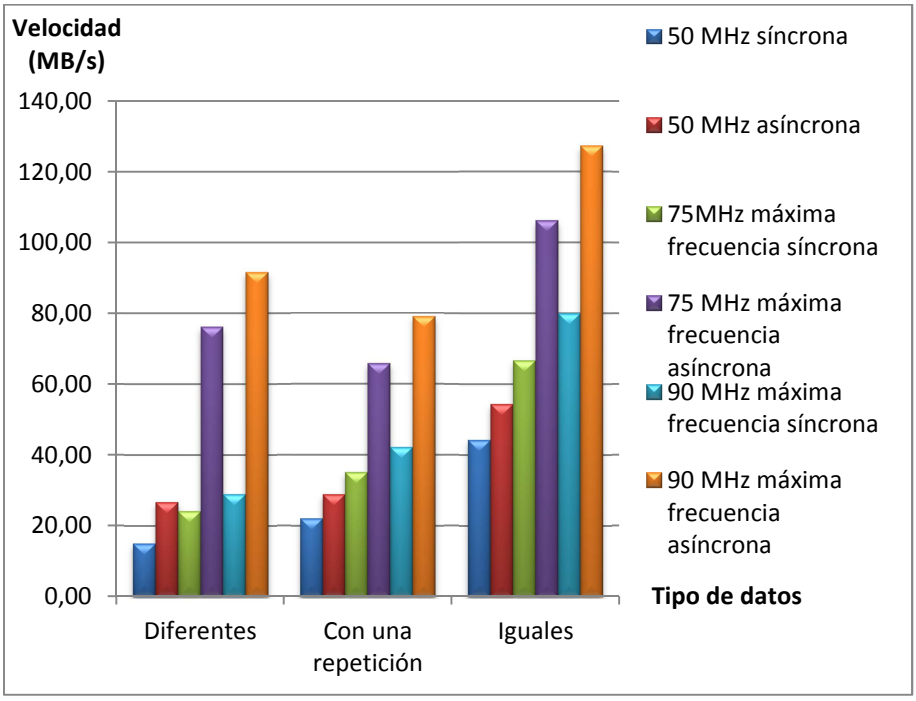


Figura 29: Comparativa velocidad del descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Spartan 3.

De la prueba realizada, en la Tabla 13 y la Figura 29 se puede comprobar que a mismas frecuencias la velocidad es igual entre la placa Nexys 2 y Spartan 3 con lo que se corrobora lo dicho anteriormente.

Por otra parte la Spartan 3 ofrece un rango mayor de frecuencias por lo que permite aumentar la velocidad máxima de transmisión un 20% (ya que la frecuencia máxima de Microblaze también aumenta un 20%).

También se ha comprobado que el tiempo de descompresión de datos es lineal con respecto al número de datos que se descomprimen, dicho en otras palabras, la velocidad de descompresión permanece constante.

Tipo de datos	610 MB		5,96 GB		11,92 GB	
	Tiempo (seg)	Rend (MB/s)	Tiempo (seg)	Rend (MB/s)	Tiempo (seg)	Rend (MB/s)
Diferentes	40	15,26	404	15,11	806	15,15
Con una repetición	27	22,61	275	22,19	551	22,15
Iguales	13	46,95	137	44,55	275	44,39

Tabla 14: Comparativa velocidad del descompresor con distinta cantidad de datos.

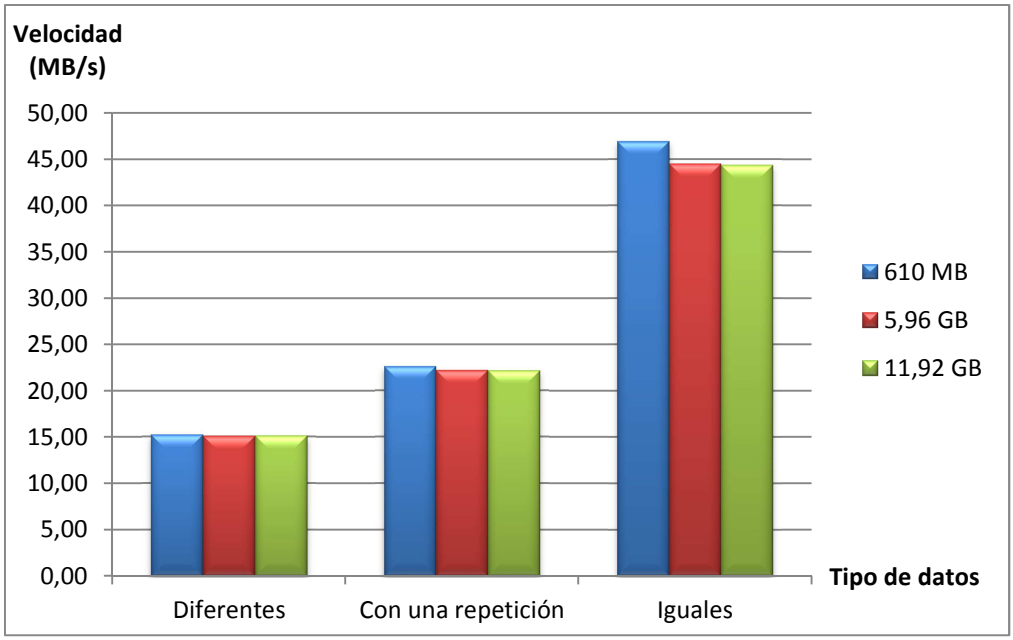


Figura 30: Comparativa velocidad del descompresor con distinta cantidad de datos.

Como se puede observar en la Tabla 14 y la Figura 30 la velocidad se mantiene constante para distinto número de datos a descomprimir. Las pruebas realizadas se basan repetir una gran cantidad de veces un envío de 16 datos, la de 610 MB corresponde a 160 millones de repeticiones, la de 5.96 GB a 1.600 millones de repeticiones y la de 11.92 GB a 3.200 millones de repeticiones. Las diferencias que se observan son despreciables y son debidas al error cometido con el cronómetro, por eso **todas las pruebas que se muestran en esta memoria han tenido un tiempo de duración mayor a 10 minutos**, pero para mostrar los resultados se han extrapolado dichos resultados a envíos de 5.96 GB para observar con mayor facilidad la diferencia de tiempos. Es decir, no todas las pruebas se han realizado con 5.96 GB de datos, sino que las más lentas, como el caso de todos los datos diferentes, si se han realizado con ese número de datos, pero las pruebas más rápidas como los datos con una repetición o todos los datos iguales se ha realizado con mayor cantidad de datos para que el tiempo de descompresión superara los 10 minutos y minimizar el error del cronómetro. Una vez obtenidos los resultados se han calculado los tiempos que tardarían dichos diseños en descomprimir 5.96 GB de datos, así es más fácil visualizar la diferencia entre las velocidades según el distinto tipo de tramas que se envían y los resultados son iguales ya que la velocidad permanece constante aunque se varíe el número de datos que se procesan. Esto se ha realizado en todas las pruebas que se han mostrado y las siguientes.

5.3. ANÁLISIS DE CONSUMO DE ENERGÍA DE LA ARQUITECTURA DE DESCOMPRESIÓN EN DISTINTAS FPGAs

También se han realizado pruebas de consumo de energía de las FPGAS. Dichas pruebas complementan a las pruebas de velocidad realizadas anteriormente ya que tan interesante es el la velocidad de descompresión de los datos como el consumo de las FPGAs ya que es posible que si el incremento de consumo es alto, no justifique el uso de un diseño que sea más rápido.

El método de medida de consumo de energía de las FPGAs ha sido usando un vatímetro. Dicho vatímetro ha sido conectado a la red eléctrica y posteriormente la FPGA ha sido conectada al vatímetro por lo que dicho elemento mide la corriente que lo atraviesa y midiendo la tensión de la red sabe la potencia total consumida por la FPGA.

El vatímetro usado ha sido "**Watts up?**" .Net. Es un dispositivo ideal para la monitorización remota. El servidor web integrado permite acceder a los datos a través de internet. Conectando el medidor a la red, el medidor automáticamente envía los datos. Con éste dispositivo es posible ver los datos en tiempo real o en formato de tabla o gráfica, configurando el tiempo de muestreo. Con este dispositivo se pueden medir valores de voltaje y corriente en un tiempo de respuesta rápido, así se puede observar el pico de energía cuando se enciende un aparato por primera vez. Los valores máximos son almacenados en memoria para su posterior visualización [WUM].

Figura 31: Vatímetro Watts up? .Net.

Como se ha descrito en las especificaciones el vatímetro se puede conectar al ordenador y guarda los datos para dibujarlos en una gráfica. En este caso no ha sido de interés porque se ha observado que los valores de consumo de energía permanecen constantes durante todo el tiempo de descompresión.

5.3.1. FPGA Nexys 2

Para usar este aparato ha sido necesario configurar la placa Nexys 2 ya que la alimentación se realiza por el mismo puerto por el que se mandan los datos, por USB, por lo que ha habido que cambiar la configuración para alimentarlo por un transformador enchufado al vatímetro.

La alimentación de la Nexys 2 tiene que tener un valor entre 5 VDC y 15 VDC con el enchufe de alimentación con clavija de 2.1 mm. Para ello hay que cambiar el pin de alimentación por USB, a alimentación WALL [Digilent-Nexys2].

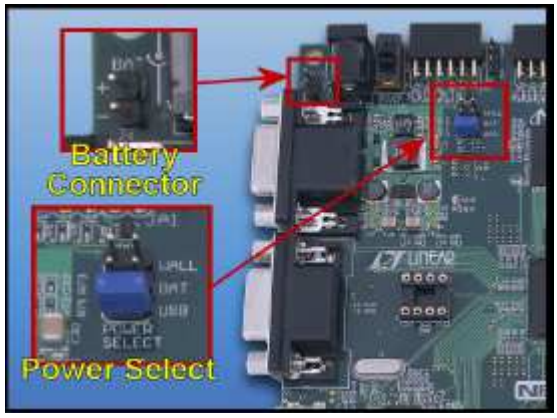


Figura 32: Modificaciones para conectar la Nexys 2 a la alimentación a la red eléctrica.

Una vez hechas estas modificaciones ya se puede conectar la FPGA a la red eléctrica por lo que se puede medir su consumo eléctrico mediante el método anteriormente descrito. Para realizar las pruebas se ha conectado el transformador de la FPGA al vatímetro y de esta forma medir la energía consumida. En las pruebas realizadas se ha observado que el consumo permanece constante con una pequeña oscilación de 100 mW, para los datos mostrados se ha realizado la media entre el valor máximo y mínimo, es decir el valor del cual oscila.

En primer lugar se ha comprobado el consumo de la placa en un único diseño. Se ha realizado en varios diseños pero en este caso se mostrará la configuración por defecto, es decir, el Microblaze configurado a 50 MHz. Se va a comparar el consumo entre distintos tipos de datos y comunicación síncrona y asíncrona:

16 datos repetidos 100 millones de veces - 5.96 GB (50 MHz)		
Tipo de datos	**Comunicación síncrona**	**Comunicación asíncrona**
	Consumo (W)	**Consumo (W)**
Diferentes	1,40	1,40
Con una repetición	1,40	1,40
Iguales	1,40	1,40

Tabla 15: Consumo energía del descompresor con distintos tipos de datos y diferente comunicación.

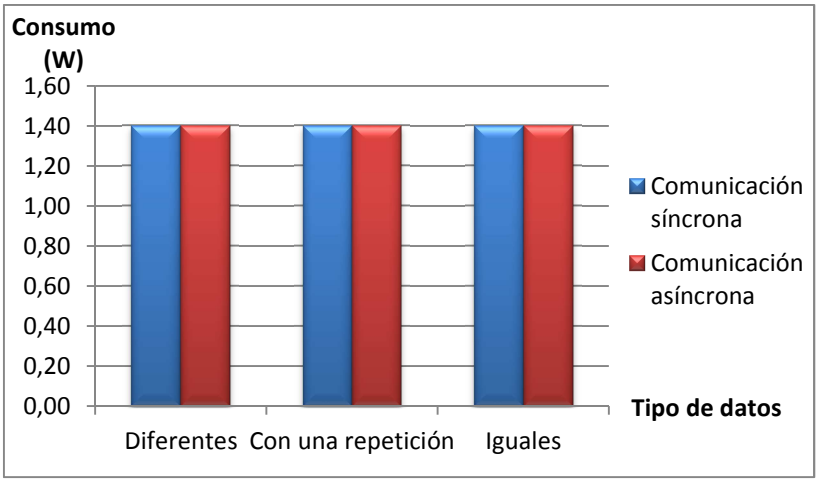


Figura 33: Consumo energía del descompresor con distintos tipos de datos y diferente comunicación.

De esta prueba en la Tabla 15 y la Figura 33 se puede deducir que el consumo del diseño permanece constante aunque se cambie el modo de comunicación (de síncrono a asíncrono) por lo que en este apartado no se hará distinción en qué tipo de comunicación se ha utilizado y tampoco depende del tipo de datos que se estén procesando.

Por otra parte se ha observado que el consumo permanece constante durante toda la descompresión, por lo que es independiente del número de datos que haya que comprimir, por lo que, para las próximas pruebas se cogerá un tamaño de número de datos pero se puede extender a descompresiones más cortas o más largas.

La primera conclusión resulta interesante, ya que el aumento de velocidad obtenido al realizar comunicación asíncrona no repercute en el consumo.

A continuación se va a mostrar los resultados obtenidos de la variación del consumo dependiendo de la frecuencia de Microblaze seleccionado:

Frecuencia (MHz)	Consumo (W)
50	1,40
50 máxima frecuencia	1,45
75 máxima frecuencia	1,60

Tabla 16: Comparación consumo energía del descompresor a distintas frecuencias Microblaze.

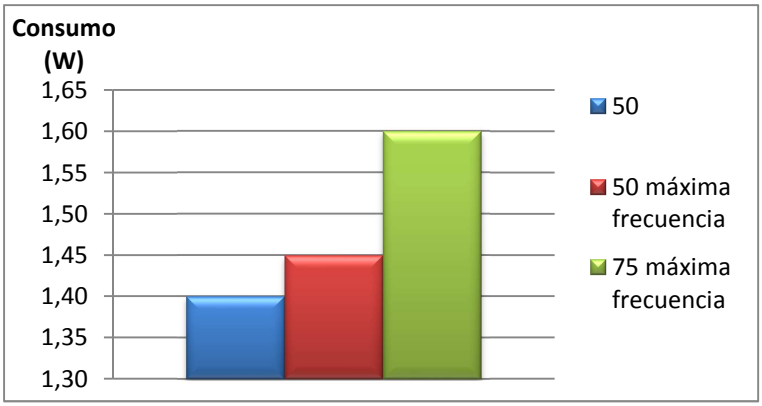


Figura 34: Comparación consumo energía del descompresor a distintas frecuencias de Microblaze.

Como es de esperar al aumentar la frecuencia de Microblaze, también aumenta el consumo, ya que se ejecutan más operaciones por segundo. La diferencia entre la configuración por defecto de Microblaze y la configuración máxima frecuencia a 50 MHz es de un 3.6% más en el consumo de energía, es despreciable dado que la frecuencia sigue siendo la misma por lo que el consumo debería ser igual. Es valor no es muy representativo dado que el aumento de velocidad en dichas circunstancias también es despreciable entre un 1% y un 7% (dependiendo del tipo de datos que se descompriman).

Por otra parte el incremento de consumo de energía al pasar del diseño por defecto al diseño de 75 MHz máxima frecuencia es del 14%, un valor aceptable si se tiene en cuenta que la frecuencia aumenta un 50%. También es un buen dato dado que el aumento de velocidad en dichas condiciones oscila entre un 50% y un 60% (dependiendo del tipo de datos que se descompriman). Por lo tanto es interesante comprobar el número de datos que se procesan por vatio, para este análisis se ha cogido el caso de que los datos se repiten una vez y comunicación asíncrona, por lo tanto:

Datos con una repetición			
Frecuencia (MHz)	**Rend (MB/s)**	**Consumo (W)**	**Rend ((MB/s)/W)**
50	22,19	1,40	15,85
50 máxima frecuencia	23,48	1,45	16,19
75 máxima frecuencia	35,22	1,60	22,01

Tabla 17: Datos procesados por vatio en la FPGA Nexys 2.

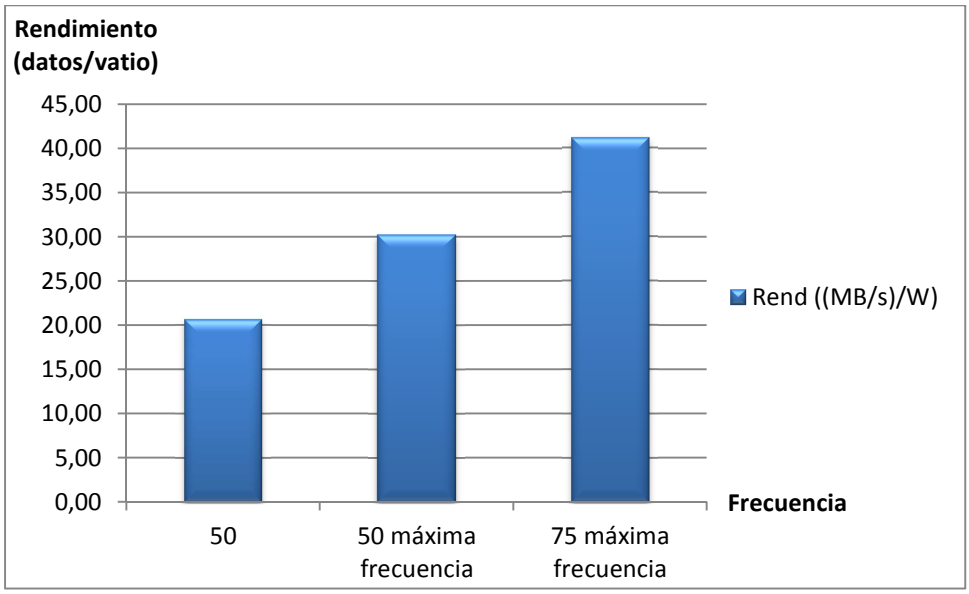


Figura 35: Datos procesados por vatio en la FPGA Nexys 2.

Como se puede observar en la Tabla 17 sabiendo el número de datos que se procesan por segundo y el consumo energético es posible saber el número de datos que se descomprimen por vatio, lo que da una mejor idea del rendimiento, tanto en velocidad como energético del diseño a distintas frecuencias, ya que como se aprecia en la Figura 35 aunque el consumo a 75 MHz es superior, la mejora en la velocidad supera dicho incremento. Por lo tanto todos los métodos vistos anteriormente para el aumento de la velocidad de descompresión son útiles. Aunque para algunas actividades en las que el consumo sea un factor crítico es posible que no se puedan obtener dichas ventajas.

5.3.2. FPGA Spartan 3

Al igual que la placa Nexys 2, con la Spartan 3 se han realizado las mismas pruebas para comprobar la variación de consumo energético en distintas condiciones. Como ya se ha comprobado el consumo permanece constante aunque varíe el número de datos que descomprimen, y el tipo de datos (si tiene muchas repeticiones o pocas). Esto ya se ha visto en el apartado anterior, experimentalmente se ha vuelto a comprobar por si en esta placa era diferente pero el resultado ha sido el mismo, pero para no repetir no se va a mostrar tal prueba.

Por lo tanto la única prueba de interés en este capítulo es la comparativa de energía consumida a distintas frecuencias de Microblaze y la diferencia de consumo entre la placa Nexys2 y Spartan 3:

Nexys 2		**Spartan 3**	
Frecuencia (MHz)	**Consumo (W)**	**Frecuencia (MHz)**	**Consumo (W)**
50	1,40	50	1,15
50 máxima frecuencia	1,45	50 máxima frecuencia	1,15
75 máxima frecuencia	1,60	75 máxima frecuencia	1,25
		90 máxima frecuencia	1,4

Tabla 18: Comparación Nexys 2 y Spartan 3 del descompresor a distintas frecuencias de Microblaze.

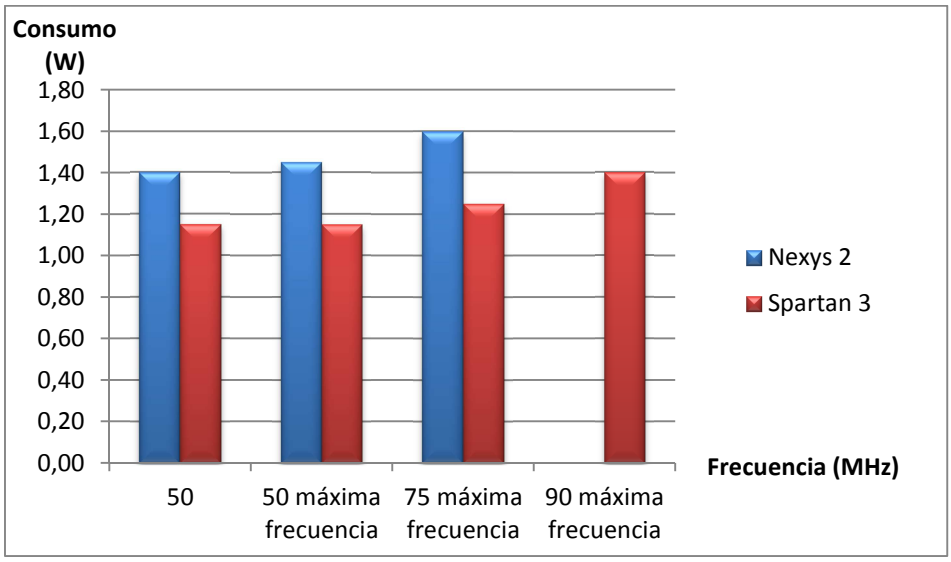


Figura 36: Comparación consumo Nexys 2 y Spartan 3 del descompresor a distintas frecuencias de Microblaze.

En primer lugar en la Tabla 18 y la Figura 36 se observa que el consumo de la Spartan 3 es inferior al de la Nexys 2, en torno a un 24% menos, una cifra bastante significativa.

Por otra parte como es de esperar el consumo cambia con la frecuencia, es decir a la misma frecuencia el consumo es el mismo ya sea un diseño por defecto o de máxima frecuencia, en este caso habrá diferencias en el coste hardware pero no en el consumo de energía. El aumento de consumo al variar entre 50 MHz y 75 MHz es del 8,7% que es incluso menos que en la Nexys 2 mientras que el aumento de la velocidad es la misma para ambas placas. Por último al aumentar a 90 MHz el incremento de consumo es de un 22%, un valor ciertamente bajo teniendo en cuenta que casi se está duplicando el valor de frecuencia y que el beneficio de velocidad oscila entre 134% y el 242% (dependiendo del tipo de datos que se descompriman. Por lo tanto se puede obtener un aumento de rendimiento muy significativo ya que la velocidad aumenta al más del doble y el consumo ni si quiera aumenta en un cuarto su valor del diseño por defecto. Se pueden alcanzar velocidades realmente altas con una penalización en consumo bastante moderada. De nuevo es interesante observar los datos procesados por vatio:

Datos con una repetición						
Frecuencia (MHz)	Nexys 2			Spartan 3		
	Rend (MB/s)	Consumo (W)	Rend ((MB/s)/W)	Rend (MB/s)	Consumo (W)	Rend ((MB/s)/W)
50	28,93	1,40	20,66	28,93	1,15	25,16
50 (MF)	43,91	1,45	30,28			
75 (MF)	65,87	1,60	41,17	65,86	1,25	52,69
90 (MF)				79,09	1,4	56,49

Tabla 19: Datos procesados por vatio en las FPGAs Nexys 2 y Spartan 3.

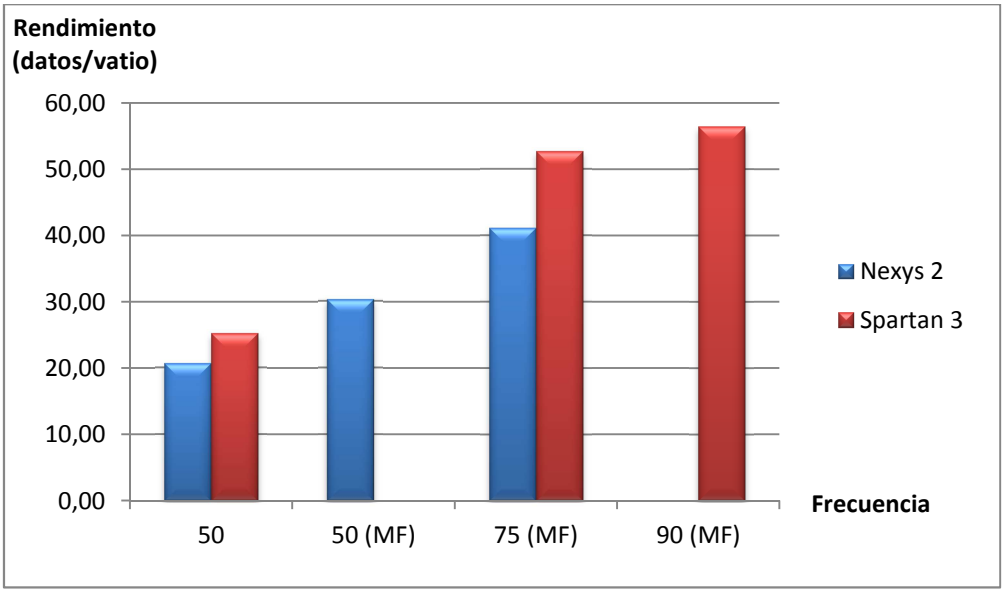


Figura 37: Datos procesados por vatio en las FPGAs Nexys 2 y Spartan 3.

Como se puede observar en la

Datos con una repetición						
Frecuencia (MHz)	Nexys 2			Spartan 3		
	Rend (MB/s)	Consumo (W)	Rend ((MB/s)/W)	Rend (MB/s)	Consumo (W)	Rend ((MB/s)/W)
50	28,93	1,40	20,66	28,93	1,15	25,16
50 (MF)	43,91	1,45	30,28			

75 (MF)	65,87	1,60	41,17	65,86	1,25	52,69
90 (MF)				79,09	1,4	56,49

Tabla **19** y la Figura 37, la FPGA Spartan 3 tiene un mayor rendimiento ya que a la misma frecuencia procesa más datos con el mismo consumo energético, a parte tiene la posibilidad de funcionar 90 MHz, en cuyo caso el rendimiento aún es mayor, en este caso también se ha realizado esta prueba con datos que se repiten una vez y comunicación asíncrona.

Como se ha podido observar en estas pruebas, tanto en la Nexys 2 como la Spartan 3, es que el componente que limita la velocidad del diseño es la comunicación, dado que al aumentar la frecuencia de Microblaze y al usar una configuración de comunicación asíncrona la velocidad de descompresión sigue aumentando, como la frecuencia de reloj del código VHDL permanece constante en todo momento esto quiere decir que en ningún momento el diseño está limitado por la velocidad del código implementado ya que en ese caso aunque se subiera la frecuencia de comunicación la velocidad de descompresión permanecería constante.

5.4. ANÁLISIS DE VELOCIDAD DE LA ARQUITECTURA COMPLETA EN DISTINTAS FPGAS

Una vez se han realizado las pruebas de velocidad y consumo de energía para el diseño de descompresión se va a proceder a mostrar las mismas pruebas realizadas al diseño completo, es decir compresor y descompresor implementados en paralelo en el que se envían los datos al compresor y una vez estos son comprimidos se envían dichos resultados al descompresor para reconstruir los datos originales. Esta arquitectura ya ha sido explicada en el capítulo 4.5. por lo que se va a proceder directamente a mostrar los resultados obtenidos.

En este caso se han realizado pruebas en distintas placas con diferentes FPGAs, dichas placas son la Nexys 2, Nexys 3, y Atlys. La Spartan 3 no se ha vuelto a probar dado que los valores obtenidos eran los mismos que en la Nexys 2 ya que la FPGA era de la misma familia y se ha preferido hacer pruebas con FPGAs distintas.

5.4.1. FPGA Nexys 2

Las pruebas realizadas son exactamente las mismas que para el descompresor, lo único que varía es que ahora hay una primera etapa de compresión y posteriormente una etapa de compresión. Lógicamente este diseño será más lento ya que existen el doble de etapas y la comunicación entre ambas.

Como ya se han hecho anteriormente un conjunto de pruebas no se van a repetir las que no aportan información, como la diferencia entre el primer y segundo diseño en el que se utilizan distintos buses FSL para transmitir los datos, dado que la velocidad y el rendimiento energético son iguales. Por lo tanto se van a mostrar las pruebas realizadas que aportan información interesante, esto será así a partir en todas las pruebas realizadas a partir de este punto. Estas pruebas son la diferencia en la velocidad de compresión – descompresión utilizando comunicación síncrona y asíncrona y con distintos valores de frecuencia de Microblaze, en este caso la de defecto (50 MHz) y el valor máximo (75 MHz máxima frecuencia), como la diferencia con 50 MHz máxima frecuencia no tenía apenas diferencias con la configuración por defecto se va a obviar.

16 datos repetidos 100 millones de veces - 5.96 GB				
Tipo de datos	**50 MHz síncrona**		**75MHz máxima frecuencia síncrona**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	1330	4,59	804	7,59
Con una repetición	870	7,02	525	11,63
Iguales	460	13,27	322	18,96
Tipo de datos	**50 MHz asíncrona**		**75 MHz máxima frecuencia asíncrona**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	1288	4,74	698	8,74
Con una repetición	856	7,13	464	13,15
Iguales	460	13,27	322	18,96

Tabla 20: Comparativa velocidad del compresor - descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Nexys 2.

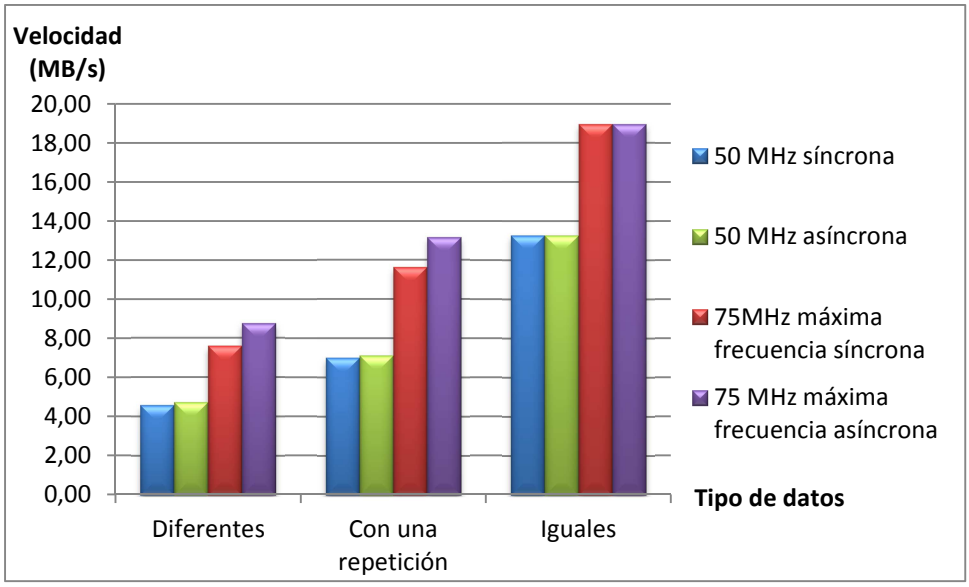


Figura 38: Comparativa velocidad del compresor - descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Nexys 2.

Como es de esperar la velocidad es inferior en el caso de la compresión – descompresión.

Se pueden observar, en la Tabla 20 y la Figura 38, valores interesantes. El primero de ellos es que ahora apenas existen diferencias entre comunicación síncrona y asíncrona lo que significa que la parte que limita la velocidad es la correspondiente al código VHDL, ya que a mayor velocidad de transmisión de datos la velocidad de procesado apenas se incrementa, en el mejor de los casos un 13%, mientras que sólo con el descompresor se obtenían mejoras muy superiores. Es difícil determinar si es el compresor o el descompresor el que limita la velocidad de procesado de datos pero, dado que la velocidad de descompresión con esa misma configuración es en torno a 6 veces superior a la del diseño conjunto es posible que la etapa de compresión sea el elemento que limita la velocidad, pero no se puede afirmar con seguridad dado que se carece de la información de la velocidad de compresión.

Por lo tanto en este caso si se precisa de una mayor rapidez de procesado es necesario aumentar la frecuencia de Microblaze ya que es la única manera en la que se obtiene una mejora notable, entre un 43% y un 65% (dependiendo del tipo de datos que se envíen). Es un dato acorde con el incremento en un 50% de la frecuencia de Microblaze.

5.4.2. FPGA Nexys 3

Esta placa no se ha utilizado en las pruebas de descompresión. Se ha decidido introducirla en este caso dado que la FPGA que contiene es una Spartan 6, un modelo superior al que utiliza la Nexys 2, para comparar si existe alguna diferencia entre el uso de una familia de FPGAs u otra.

Al ser un modelo superior la principal ventaja es que la frecuencia de reloj del código VHDL pasa de ser 50 MHz a ser 100 MHz, es decir se duplica, por lo que se va a poder comprobar si el elemento que limita el diseño es el código VHDL o la comunicación entre bloques.

De igual manera se va a proceder a hacer una prueba de velocidad de compresión descompresión a diferentes frecuencias de Microblaze y con comunicación síncrona y asíncrona.

Hay que destacar que al tratarse de una placa distinta las frecuencias de Microblaze que se pueden configurar son distintas, en este caso se ha elegido una frecuencia parecida a la de defecto de la Nexys 2 (50 MHz) que en este caso pasará a ser 66 MHz, y la frecuencia máxima que se puede configurar que es 83 MHz.

16 datos repetidos 100 millones de veces - 5.96 GB				
Tipo de datos	**66 MHz síncrona**		**83MHz máxima frecuencia síncrona**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	808	7,55	703	8,68
Con una repetición	626	9,75	507	12,04
Iguales	436	14,00	320	19,07
Tipo de datos	**66 MHz asíncrona**		**83 MHz máxima frecuencia asíncrona**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	784	7,79	619	9,86
Con una repetición	616	9,91	444	13,75
Iguales	436	14,00	320	19,07

Tabla 21: Comparativa velocidad del compresor - descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Nexys 3.

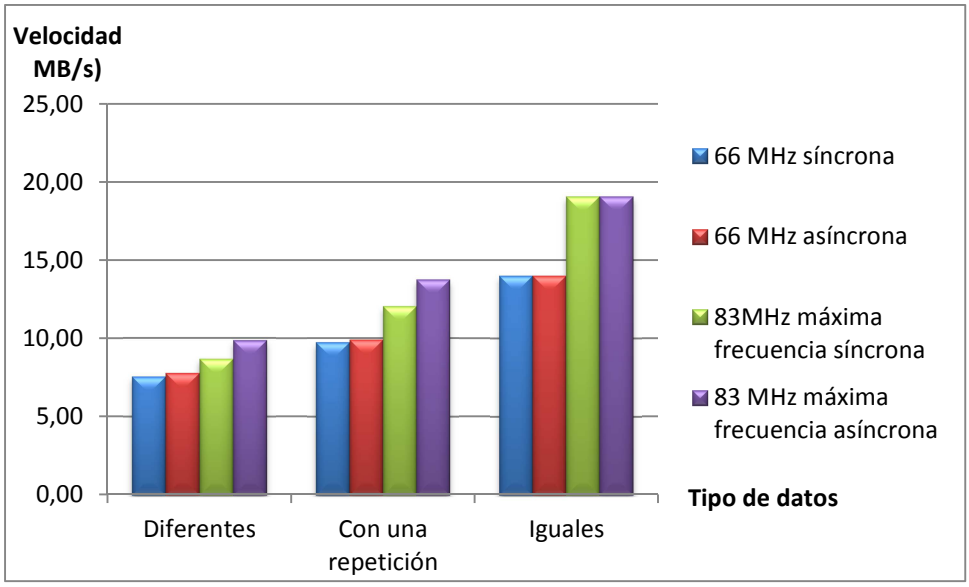


Figura 39: Comparativa velocidad del compresor - descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Nexys 3.

Los resultados obtenidos en la Tabla 21 y la Figura 39 son muy parecidos que con la Nexys 2. Según se aumenta la frecuencia aumenta la velocidad, y al pasar a comunicación asíncrona se obtiene una mejora pero poco representativa en comparación con las pruebas realizadas anteriormente al descompresor. Los datos obtenidos son coherentes ya que la frecuencia aumente un 25% y el tiempo de descompresión disminuye entre un 23% y 26% dependiendo del tipo de datos que se descompriman.

Por otra parte el beneficio obtenido entre la comunicación asíncrona y la síncrona es aproximadamente un 13%, que sigue siendo un ahorro importante de tiempo de descompresión pero es muy inferior que en el caso de la descompresión.

Por último cabe destacar que en el caso de que todos los datos sean iguales no hay diferencia entre la comunicación síncrona y asíncrona, es un dato extraño ya que el tiempo de transmisión de datos es menor en el caso de la asíncrona y la frecuencia del VHDL es el doble que con la Nexys, mientras que los datos obtenidos son similares. Esto, posiblemente, es debido a que la prueba de velocidad se hace de la siguiente manera. En primer lugar se envían los datos al compresor, éste los comprime y los devuelve a Microblaze, una vez todos son devueltos, Microblaze envía esos datos al descompresor, y una vez descomprimidos se envían de vuelta a Microblaze, es decir, que primero se realiza la compresión y hasta que no ha acabado no se comienza con la descompresión y, es probable, que ese tiempo de espera es el que limite la velocidad del diseño. Ya que queda descartado que el problema sea el código VHDL ya que al tener una frecuencia el doble que con la Nexys 2, la velocidad ha mejorado pero no al doble por lo que no es el elemento limitante, además la velocidad es mayor también debida a que la frecuencia de Microblaze es mayor, 66 MHz para la Nexys 3 y 50 MHz para la Nexys 2.

5.4.3. FPGA Atlys

Para acabar con las pruebas de velocidad en distintos dispositivos se ha elegido una placa Atlys, mucho más potente que las utilizadas hasta ahora. La FPGA que incluye esta placa es de la misma familia que la de la Nexys 3, es decir una Spartan 6, y la frecuencia de operación del código VHDL también son 100 MHz. El rango de frecuencias disponibles para el Microblaze es el mismo que para la Nexys 3 por lo que se han escogido las mismas frecuencias.

Esta placa es mucho más potente que la Nexys 3 ya que los dispositivos para controlar entradas y salidas de datos son mucho más sofisticados (incluye varias entradas y salidas HDMI) pero para el uso que se va a dar en este caso, lo que importa es la FPGA y las frecuencias de trabajo por lo que para este proyecto es prácticamente igual que la Nexys 3. Se ha escogido esta placa porque incluye un circuito propio para la medida de consumo energético lo cual será útil en el próximo apartado.

Los datos obtenidos en las pruebas realizadas son los siguientes:

16 datos repetidos 100 millones de veces - 5.96 GB				
Tipo de datos	**66 MHz síncrona**		**83MHz máxima frecuencia síncrona**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	794	7,69	696	8,77
Con una repetición	616	9,91	547	11,16
Iguales	436	14,00	320	19,07
Tipo de datos	**66 MHz asíncrona**		**83 MHz máxima frecuencia asíncrona**	
	Tiempo (seg)	**Rendimiento (MB/s)**	**Tiempo (seg)**	**Rendimiento (MB/s)**
Diferentes	772	7,91	619	9,86
Con una repetición	606	10,07	444	13,75
Iguales	436	14,00	320	19,07

Tabla 22: Comparativa velocidad del compresor - descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Atlys.

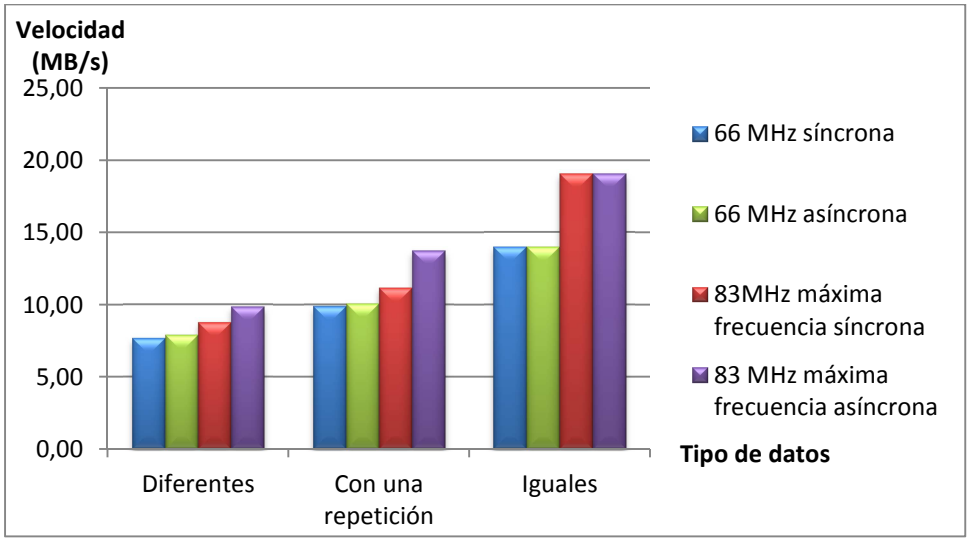


Figura 40: Comparativa velocidad del compresor - descompresor a distintas frecuencias con comunicación síncrona y asíncrona en Atlys.

Como es de esperar los resultados de la Tabla 22 y la Figura 40 son prácticamente iguales que con la Nexys 3 por lo que no da lugar a un análisis demasiado exhaustivo. Como es lógico al aumentar la frecuencia de Microblaze aumenta la velocidad y con comunicación asíncrona el diseño es algo más rápido excepto que todos los datos sean iguales, en cuyo caso es igual.

Ahora parece interesante hacer una comparativa entre las tres FPGAs para poder hacer una mejor valoración en el posterior capítulo de consumo energético, para hacer dicha comparativa se va a escoger una frecuencia de Microblaze de 83 MHz en el caso de la Nexys 3 y Atlys y 75 MHz y comunicación asíncrona para la Nexys 2.

16 datos repetidos 100 millones de veces - 5.96 GB						
Tipo de datos	Nexys 2 (75 MHz)		Nexys 3 (83 MHz)		Atlys (83 MHz)	
	Tiempo (seg)	Rend (MB/s)	Tiempo (seg)	Rend (MB/s)	Tiempo (seg)	Rend (MB/s)
Diferentes	698	8,74	619	9,86	619	9,86
Con una repetición	464	13,15	444	13,75	444	13,75
Iguales	322	18,96	320	19,07	320	19,07

Tabla 23: Comparativa velocidad del compresor - descompresor en distintas FPGAs.

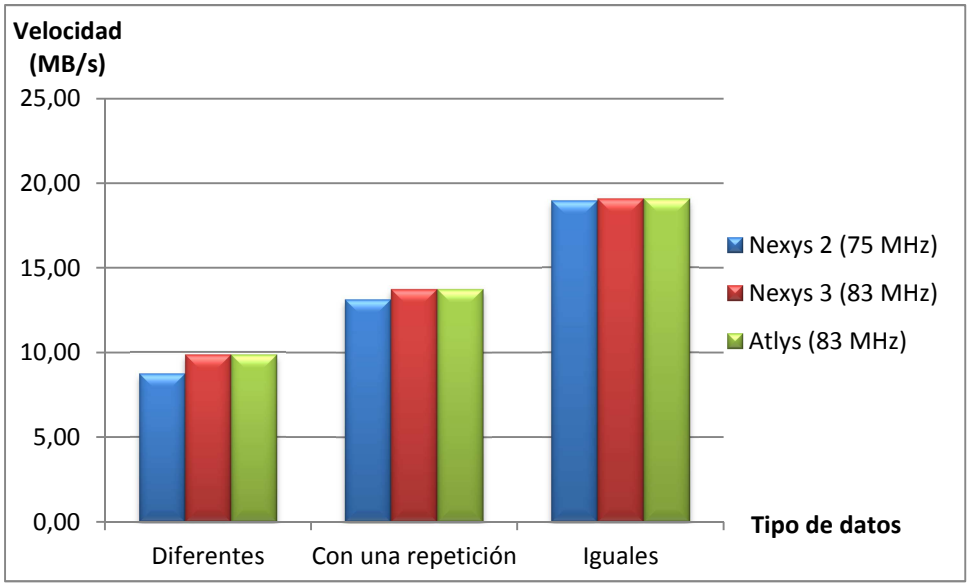


Figura 41: Comparativa velocidad del compresor - descompresor en distintas FPGAs.

Como se puede apreciar en la Figura 41 no se aprecia una diferencia notable entre el uso de una placa u otra, lo que quiere decir que no hay apenas diferencia entre que el código VHDL funcione a 50 MHz o a 100 MHz. La diferencia que existe entre la primera placa y las otras dos es en torno al 10% que es la misma diferencia que hay entre la frecuencia que está configurado el Microblaze.

5.5. ANÁLISIS DE CONSUMO DE ENERGÍA DE LA ARQUITECTURA COMPLETA EN DISTINTAS FPGAS

Para terminar con las pruebas realizadas en distintas FPGAs se va a hacer una evaluación de rendimiento energético correspondiente a las pruebas de velocidad realizadas en el capítulo anterior. Por último se mostrará una comparación de las tres FPGAS utilizadas.

5.5.1. FPGA Nexys 2

A continuación se va a proceder a mostrar las pruebas realizadas con la placa Nexys 2. De igual manera que se realizó anteriormente hay que cambiar la configuración de dicha placa para alimentarla mediante un transformador enchufado al vatímetro ya que el método de medida es el mismo que en la anterior ocasión.

De igual manera como ya se ha experimentado que el consumo no varía ni con el número de datos que se procesan, ni con el tipo de datos que se envían (si son datos que se repiten o no), ni si se está utilizando una comunicación síncrona o asíncrona. Por lo tanto se van a obviar estas pruebas, por lo que la única prueba de interés es observar como varía el consumo en función de las frecuencias utilizadas.

Frecuencia (MHz)	Consumo (W)
50	1,45
75 máxima frecuencia	1,60

Tabla 24: Comparación consumo energía del compresor - descompresor a distintas frecuencias Microblaze con la placa Nexys 2.

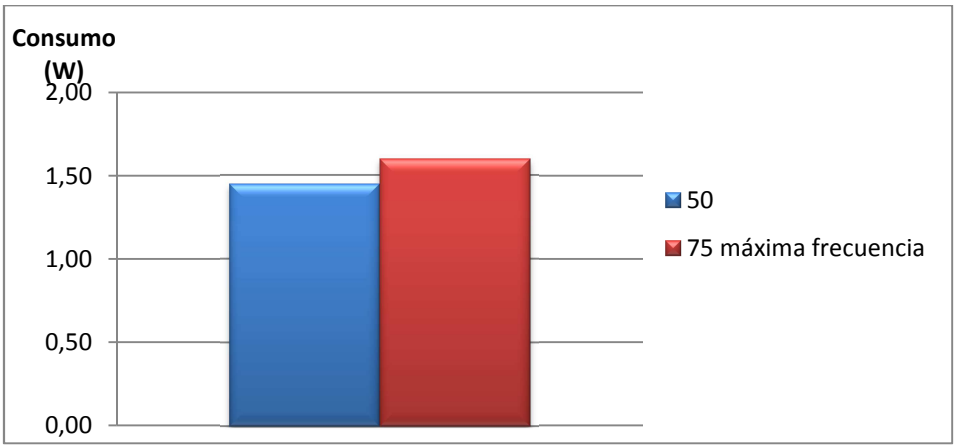


Figura 42: Comparación consumo energía del compresor - descompresor a distintas frecuencias Microblaze con la placa Nexys 2.

En este, en la Tabla 24 caso el consumo es un 4% mayor que en el caso de la descompresión, es normal que consuma más dado que es un diseño de mayor envergadura. Al aumentar la frecuencia un 50% se obtiene un aumento en el consumo de un 10%. Es un valor bastante bueno dado que el incremento de velocidad al cambiar de una frecuencia a otra oscila entre el 43% y el 85%, dependiendo del tipo de datos que se procesen, por lo que se puede obtener una buena mejora a costa de un poco más de consumo energético.

5.5.2. FPGA Nexys 3

No se tiene referencia anterior del consumo de esta placa pero dado que utiliza una frecuencia de código VHDL que duplica a la de la Nexys 2 y la frecuencia de Microblaze también es algo mayor es de esperar que el consumo también sea algo más elevado. Por otra parte también es posible pensar que al ser una placa más potente tenga mejores componentes y esté mejor optimizada por lo que es posible que consuma menos.

Frecuencia (MHz)	Consumo (W)
66	1,30
83 máxima frecuencia	1,35

Tabla 25: Comparación consumo energía del compresor - descompresor a distintas frecuencias Microblaze con la placa Nexys 3.

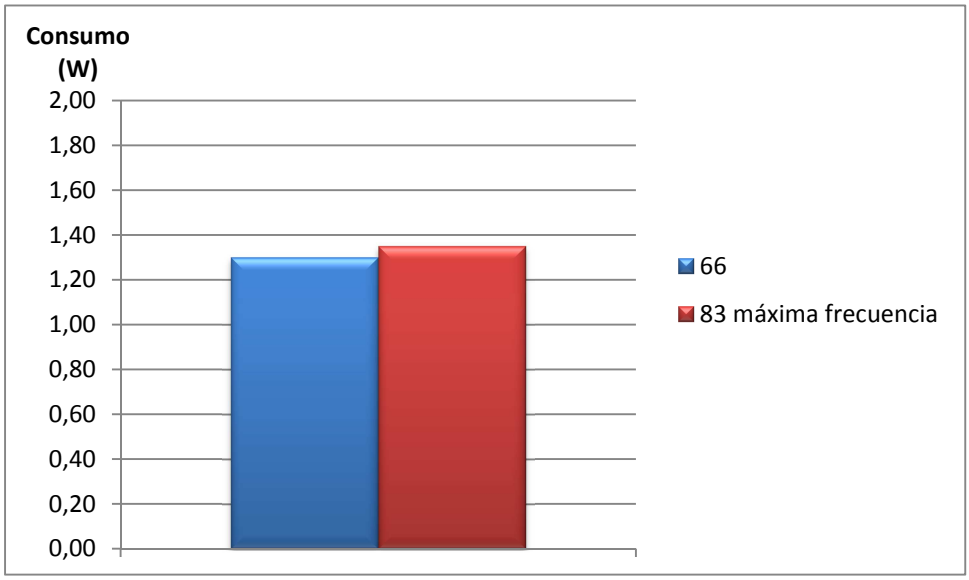


Figura 43: Comparación consumo energía del compresor - descompresor a distintas frecuencias Microblaze con la placa Nexys 3.

Sorprendentemente como antes se ha comentado esta placa tiene un consumo menor, aunque al ser más potente y disponer de más recursos es de esperar lo contario.

En la Tabla 25 se puede observar un aumento de la frecuencia es de un 25%. En cambio el aumento en el consumo es de un 4%. Por otro lado el aumento en la velocidad al cambiar la frecuencia varía entre un 26% y 36%. Es lógico que sea menor que en la Nexys 2 dado que la variación de frecuencia también es menor.

Por el momento con ambas dos placas se puede obtener una mejora considerable de velocidad de procesamiento a costa de una pequeña penalización en el consumo.

5.5.3. FPGA Atlys

En último lugar se va a mostrar los datos obtenidos con la placa Atlys. Esta placa tiene un circuito integrado que se encarga de medir la potencia consumida por la placa. Es de esperar que este método sea más fiable que el usado en los demás apartados ya que esta todo integrado en la misma placa y tiene un programa que monitoriza el consumo, los valores se pueden almacenar y dicho programa es capaz de modelar gráficas con los datos obtenidos, no es de interés mostrar estas graficas ya que se comprobó que el consumo era constante en el tiempo y el único interés es el valor del consumo. Se ha decidido hacer una medición doble, es decir con el método anterior y con el programa de la placa, de esta manera se puede corroborar si los datos anteriormente obtenidos son fiables o por el contrario existe alguna discrepancia entre los métodos de medida.

La medida de energía proporciona una alta precisión (superior al 1%). Mide en tiempo real la corriente y potencia de la fuente de alimentación [Digilent-Atlys].

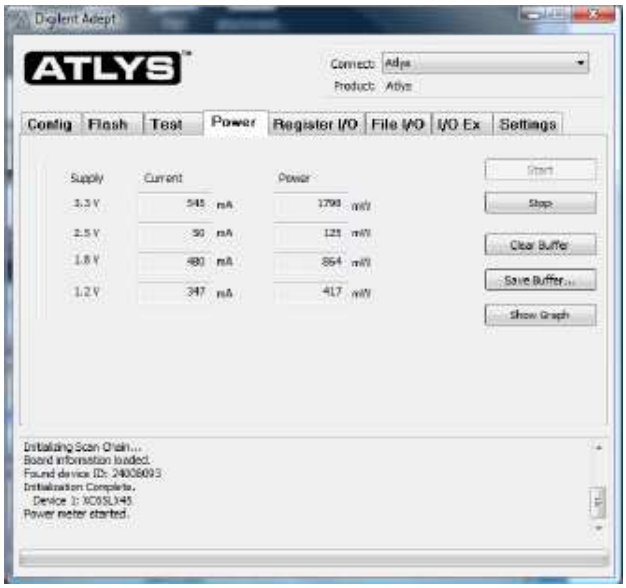


Figura 44: Visualización interface monitorización consumo energético Atlys.

Es de esperar que el consumo de esta placa sea mayor, ya que es una placa notablemente más potente que las usadas hasta ahora que permite hacer diseños complejos con entradas y salidas de última generación. La FPGA lleva un disipador integrado por lo que se calienta más que los anteriores y esto es debido a un mayor consumo energético.

Por lo tanto se obtendrán dos medidas de potencia para un mismo caso, estas medidas son las siguientes:

	66 (MHz)	83 (MHz) máxima frecuencia
Medición	**Consumo (W)**	**Consumo (W)**
Vatímetro	4,21	4,36
Circuito Atlys	4,215	4,368

Tabla 26: Comparación consumo energía del compresor – descompresor con distinto método de medición a distintas frecuencias Microblaze con la placa Atlys.

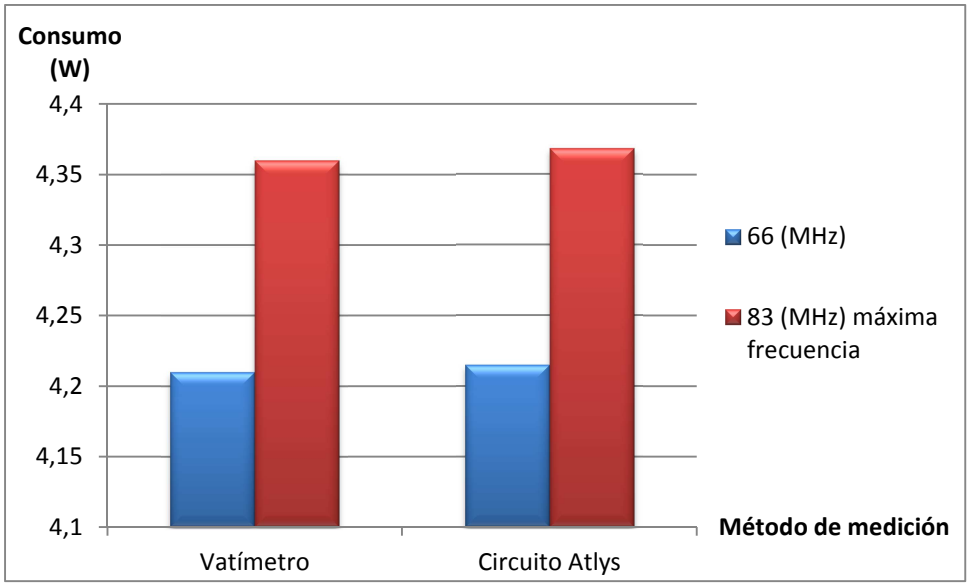


Figura 45: Comparación consumo energía del compresor – descompresor con distinto método de medición a distintas frecuencias Microblaze con la placa Atlys.

En la Tabla 26 y la Figura 45 se observa un aumento de la frecuencia es de un 25%. En cambio el aumento en el consumo es de un 3.5%. Por otro lado el aumento en la velocidad al cambiar la frecuencia varía entre un 25% y 36%. También se puede observar que el consumo es mucho más elevado, es de esperar dado que es una placa mucho más potente y está preparada para diseños de alta complejidad, pero la variación de consumo entre esas frecuencias se mantiene a la par que en la Nexys 3. Cabe destacar que los datos obtenidos por los dos métodos de medición son exactamente iguales ya que en por medio del circuito integrado en la placa el último decimal varía constantemente pudiéndose considerar despreciable.

Para finalizar con las pruebas con distintas FPGAs se va a hacer, a modo de resumen, una comparación entre ellas de velocidad y consumo a su frecuencia máxima configurable de Microblaze y escritura asíncrona.

16 datos repetidos 100 millones de veces - 5.96 GB						
Tipo de datos	Nexys 2 (75 MHz)		Nexys 3 (83 MHz)		Atlys (83 MHz)	
	Consumo (W)	Rend (MB/s)	Consumo (W)	Rend (MB/s)	Consumo (W)	Rend (MB/s)
Diferentes	1,6	8,74	1,35	9,86	4,36	9,86
Con una repetición	1,6	13,15	1,35	13,75	4,36	13,75
Iguales	1,6	18,96	1,35	19,07	4,36	19,07

Tabla 27: Comparación velocidad y consumo energético entre FPGAs.

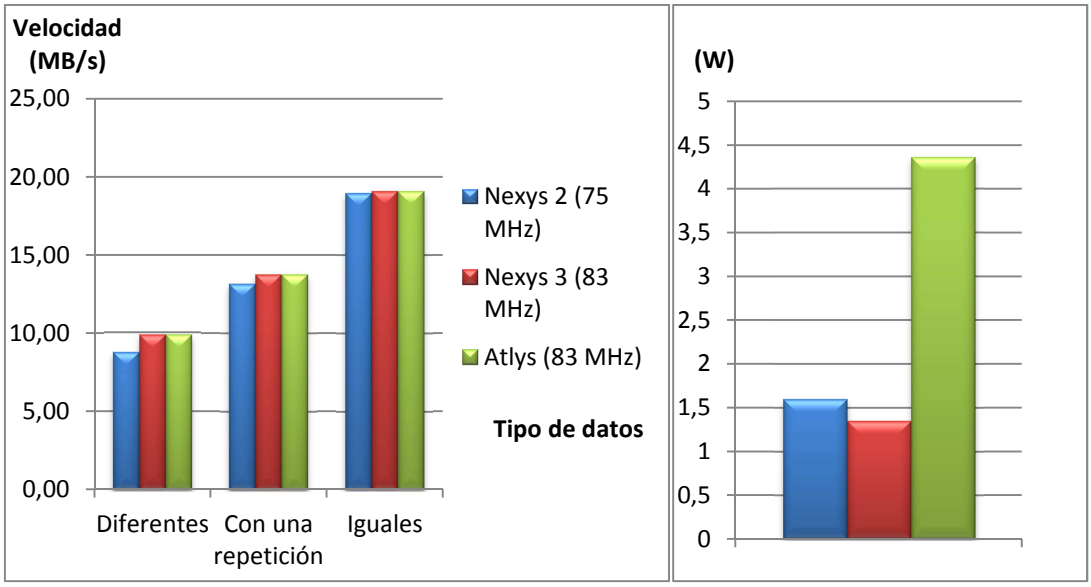


Figura 46: Comparación velocidad y consumo energético entre FPGAs.

Después de todas las pruebas realizadas se puede observar en la Tabla 27 y la Figura 46 las placas más rápidas son la Nexys 3 y la Atlys que llevan una FPGA de la familia Spartan 6. La velocidad de dichas placas oscila entre un 1% y un 13% mayor que el de la Nexys 2, ya que tiene una FPGA de la familia Spartan 3. Todos los datos de velocidad son positivos, ya que son superiores a lo esperado.

En el apartado del consumo existe una clara diferencia. La placa Atlys tiene un consumo mucho mayor al resto en torno a tres veces superior debido a que es una placa con otros fines, en otras palabras, está preparada para hacer un uso de ella mucho más exigente. En cuanto la Nexys 3 es interesante el dato del consumo ya que es menor que su predecesora la Nexys 2, que cabe esperar que ocurra como con la Atlys, que al ser una placa con más recursos esto se penalice con un consumo superior, pero en este caso es lo contrario. La Nexys 2 tiene un consumo un 18% mayor. Por lo tanto es interesante hacer una comparación entre velocidad de procesado y consumo energético.

Rendimiento distintas FPGAs									
Tipo de datos	Nexys 2 (75 MHz)			Nexys 3 (83 MHz)			Atlys (83 MHz)		
	Rend (MB/s)	Cons (W)	Rend ((MB/s)/W)	Rend (MB/s)	Cons (W)	Rend ((MB/s)/W)	Rend (MB/s)	Cons (W)	Rend ((MB/s)/W)
Diferentes	8,74	1,6	5,46	9,86	1,35	7,30	9,86	4,36	2,26
Una repetición	13,15	1,6	8,22	13,75	1,35	10,19	13,75	4,36	3,15
Iguales	18,96	1,6	11,85	19,07	1,35	14,13	19,07	4,36	4,37

Tabla 28: Datos procesados por vatio en las FPGAs Nexys 2, Spartan 3 y Atlys a máxima frecuencia.

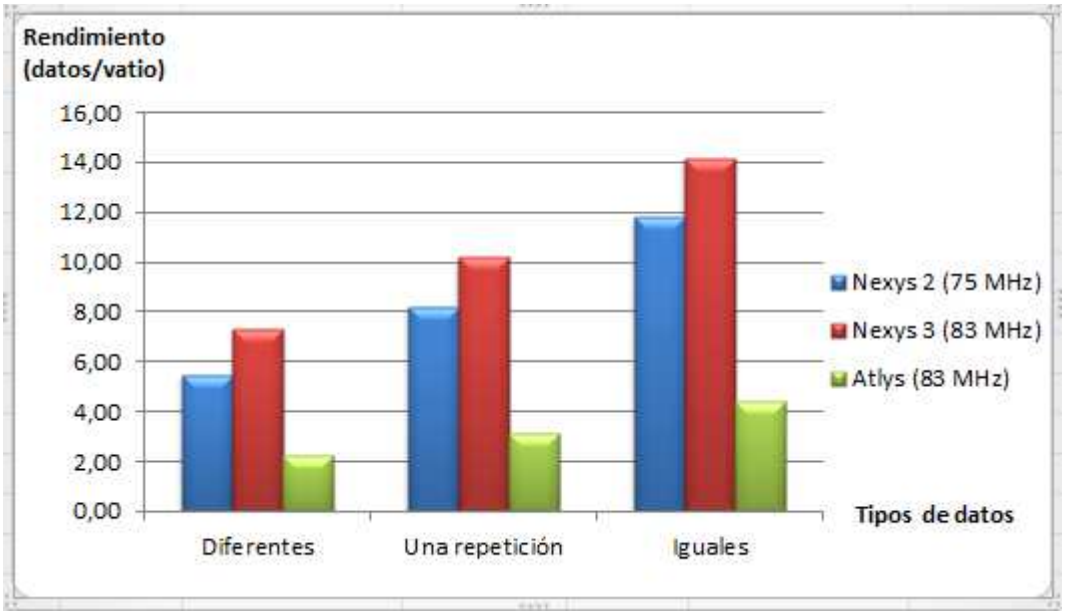


Figura 47: Datos procesados por vatio en las FPGAs Nexys 2, Spartan 3 y Atlys a máxima frecuencia.

Por lo tanto en cuanto a la velocidad todas las placas tienen un resultado similar sin grandes cambios. Y en el consumo hay una placa que claramente consume mucho más y las otras dos tienen un consumo mucho más comedido. Cabe resaltar otra vez la placa Nexys 3, como se puede observar en la Figura **47**, el rendimiento es mucho mayor que ya que es la que más velocidad de compresión – descompresión consigue un 18% más (al igual que la Atlys), con una velocidad de procesado similar a la Nexys 2 y a la vez ser la placa que menos consume un 18 % menos que la Nexys 2, por lo que el ratio datos procesados y consumo energético es mucho mejor el de la placa Nexys 3.

5.6. ANÁLISIS DE COSTE HARDWARE DE LOS DISEÑOS EN DISTINTAS FPGAS

En los anteriores apartados se han visto los resultados de múltiples pruebas de velocidad y consumo energético del diseño, pero ahora se va a realizar una comparativa de coste hardware de la FPGA. En concreto los datos interesantes son la cantidad de biestables, LUTs, bloques de E/S y DCMs, que requiere cada diseño.

Los bloques de E/S se utilizan para conectar externamente la FPGA, ofreciendo distintos voltajes de conexión y dotándola de una mayor adaptación de impedancias. También incluye buffering de E/S.

Un DCM (digital clock manager) es una función que se utiliza para la manipulación y control de señales de reloj.

5.6.1. Análisis coste hardware arquitectura de descompresión en distintas FPGAs

En primer lugar se va a realizar un estudio del coste hardware en la arquitectura de descompresión. Dicha arquitectura se ha implementado en dos placas distintas, Nexys 2 y Spartan 3.

Se va a proceder a realizar un análisis para distintas frecuencias de Microblaze en la misma placa y posteriormente un análisis entre las dos placas.

- **Descompresor 50 MHz por defecto Nexys 2:**
  - Biestables: 4.026 utilizados de 9.312 43%
  - LUTs de 4 entradas: 6.674 utilizadas de 9.312 71%
  - Bloques de E/S: 101 utilizadas de 232 43%
  - DCMs: 1 utilizados de 4 25%

- **Descompresor 50 MHz máxima frecuencia Nexys 2:**
  - Biestables: 4.331 utilizados de 9.312 46%.
  - LUTs de 4 entradas: 7.107 utilizadas de 9.312 76%
  - Bloques de E/S: 101 utilizadas de 232 14%
  - DCMs: 1 utilizados de 4 25%

- **Descompresor 75 MHz máxima frecuencia Nexys 2:**
  - Biestables: 4.577 utilizados de 9.312 49%.
  - LUTs de 4 entradas: 7.675 utilizadas de 9.312 82%
  - Bloques de E/S: 101 utilizadas de 232 14%
  - DCMs: 1 utilizados de 4 25%

Como ya se ha comentado con anterioridad, la mejora obtenida en la velocidad de descompresión es penalizada con cantidad de recursos utilizados por la FPGA.

Para todos los diseños los bloques de entrada y salida y los DCMs utilizados son los mismos, los únicos recursos que varían son los biestables y los LUTs de 4 entradas. En ninguno de los tres casos se puede destacar un aumento notable en los recursos utilizados. Al cambiar

la configuración a máxima frecuencia se obtiene un incremento del 3% en número de biestables utilizados y un 5% en LUTs de 4 entradas. En el caso de 75 MHz el incremento de biestables es del 6% y de LUTs de 4 entradas del 11%. Son buenos valores ya que el aumento de la velocidad es mucho mayor. Teniendo en cuenta que es la placa de menor categoría que se ha utilizado, y en este caso ninguno de los recursos utilizados llega a un nivel crítico, por lo que el resto de placas tendrán aún un mayor margen.

Se ha comprobado que para las distintas placas el aumento porcentual de la cantidad de recursos utilizados al cambiar la frecuencia de Microblaze es idéntico por lo que no es de interés mostrar dichos valores. Por lo tanto se va a proceder a mostrar las diferencias entre las placas Nexys 2 y Spartan 3 para la configuración por defecto de Microblaze.

- **Descompresor 50 MHz por defecto Nexys 2:**
  - Biestables: 4.026 utilizados de 9.312 43%
  - LUTs de 4 entradas: 6.674 utilizadas de 9.312 71%
  - Bloques de E/S: 101 utilizadas de 232 43%
  - DCMs: 1 utilizados de 4 25%
- **Descompresor 50 MHz por defecto Spartan 3:**
  - Biestables: 3.195 utilizados de 11.776 27%
  - LUTs de 4 entradas: 5.433 utilizadas de 11.776 46%
  - Bloques de E/S: 41 utilizadas de 372 11%
  - DCMs: 1 utilizados de 8 12%

En este caso si se observan diferencias importantes en cuanto al consumo de recursos hardware. No es posible comparar el porcentaje de recursos usados ya que la Spartan 3 posee mayor número de recursos, por ejemplo posee 11.776 biestables mientras que la Nexys 2 solo consta de 9.312. Pero cabe destacar que la Spartan 3 utiliza un 26% menos de biestables, 23% menos de LUTS de 4 entradas y por último menos de la mitad de bloques E/S. Teniendo en cuenta que la velocidad máxima de ambos diseños es la misma y que el consumo energético también es menor en la Spartan 3, quiere decir que aunque la FPGA sea de la misma familia, ésta última aparte de disponer de más recursos tiene un mejor aprovechamiento de los mismos.

5.6.2. Análisis coste hardware arquitectura completa en distintas FPGAs

En segundo lugar se ha realizado una comparación del consumo de elementos hardware de los distintos diseños para la arquitectura completa (compresor – descompresor). Al igual que en el caso anterior se va a mostrar las diferencias entre las distintas configuraciones de Microblaze en una misma placa y posteriormente las diferencias entre las distintas placas utilizadas.

- **Compresor - descompresor 50 MHz por defecto Nexys 2:**
  - Biestables: 4.305 utilizados de 9.312 46%
  - LUTs de 4 entradas: 6.644 utilizadas de 9.312 71%
  - Bloques de E/S: 101 utilizadas de 232 43%
  - DCMs: 1 utilizados de 4 25%

- **Compresor - descompresor 75 MHz máxima frecuencia Nexys 2:**
  - Biestables: 4.543 utilizados de 9.312 48%.
  - LUTs de 4 entradas: 6.979 utilizadas de 9.312 74%
  - Bloques de E/S: 101 utilizadas de 232 14%
  - DCMs: 1 utilizados de 4 25%

A primera vista es extraño que el modelo de la arquitectura completa utilice menos recursos hardware que la arquitectura de compresión. Pero hay que tener en cuenta que por problema de limitación de número de buses FSL el modelo completo sólo cuenta con dos campos por lo que tiene cuatro coprocesadores de descompresión y un único coprocesador de compresión, mientras que el modelo de descompresión cuenta con seis coprocesadores, por lo que es lógico que requiera más recursos.

En el caso de aumentar la frecuencia de Microblaze los bloques de E/S y los DCMs permanecen constantes para ambos diseños, los biestables aumentan un 2% y los LUTs de 4 entradas un 3%. El incremento de recursos utilizados es aproximadamente la mitad que en el caso de descompresión, pero también hay que tener en cuenta que en la arquitectura completa el incremento de velocidad obtenido mediante esta técnica también era notablemente inferior.

En el caso de la comparación entre distintas FPGAs, se va a proceder a hacer una comparación de igual manera que en el descompresor, es decir, utilizando la configuración por defecto de Microblaze ya que el incremento porcentual de recursos al cambiar la frecuencia permanece constante. Así pues se obtienen los siguientes resultados.

- **Compresor - descompresor 50 MHz por defecto Nexys 2:**
  - Biestables: 4.305 utilizados de 9.312 46%
  - LUTs de 4 entradas: 6.644 utilizadas de 9.312 71%
  - Bloques de E/S: 101 utilizadas de 232 43%
  - DCMs: 1 utilizados de 4 25%

- **Compresor - descompresor 50 MHz por defecto Nexys 3:**
  - Biestables: 6.390 utilizados de 18.244 35%.
  - LUTs de 4 entradas: 7.610 utilizadas de 9.112 83%
  - Bloques de E/S: 129 utilizadas de 232 55%
  - DCMs: 1 utilizados de 8 12%

- **Compresor - descompresor 50 MHz por defecto Atlys:**
  - Biestables: 5.637 utilizados de 54.576 10%.
  - LUTs de 4 entradas: 7.123 utilizadas de 27.288 26%
  - Bloques de E/S: 101 utilizadas de 218 46%
  - DCMs: 1 utilizados de 8 12%

Al igual que en la descompresión, aquí se observa una variedad significativa de los recursos utilizados.

Como en casos anteriores los bloques de E/S permanecen en valores que prácticamente se pueden considerar iguales y los DCMs utilizados de igual manera son iguales.

En este caso es curioso porque las placas más potentes consumen más recursos que las otras. Cosa que en principio es de esperar lo contario (como ocurre en el caso de la descompresión). Pero en este caso las placas son mucho más potentes, y están diseñadas para un uso avanzado, por lo que tienen más puertos de entrada. También destaca el número total de biestables y LUTs de 4 entradas que disponen la Nexys 3 y más aún la Atlys, donde se corrobora que están preparadas para diseños que requieran de muchos más recursos.

El incremento de coste de recursos de la placa Nexys 3 sobre la Nexys 2, es de un 48% para los biestables y un 14% para las LUTs de 4 entradas. En el caso de la Atlys respecto la Nexys 2 es de un 31% para los biestables y un 7%.

En cambio hay que tener en cuenta que la placa Nexys 3 ofrecía la misma velocidad que las otras dos placas con un consumo energético considerablemente menor y dado que dispone de recursos hardware más que suficientes, en caso de tener que instalar este algoritmo en una placa, la Nexys 3 sería la mejor opción ya que en todas las pruebas realizadas obtiene buenos resultados.

5.7. COMPARACIÓN ENTRE ARQUITECTURA EN VHDL, Y EN C EN MICROBLAZE Y CPU INTEL CORE I7

Para concluir con las pruebas, en último lugar se ha hecho una comparación entre tres arquitecturas distintas, la primera la arquitectura hardware implementada en VHDL que se ha descrito en este proyecto, en segundo lugar se ha diseñado un programa en lenguaje C y programado en Microblaze, es decir es un diseño software pero se ha implementado en la misma FPGA que el diseño hardware, y en tercer lugar una arquitectura software que era existente previo al proyecto. Los descompresores software utilizan el mismo algoritmo de descompresión que el hardware. El código C que se ha implementado es prácticamente igual que el hardware pero en lenguaje C, y ese es el código que se ha introducido en Microblaze, es decir, aunque sea una FPGA está ejecutando código C. Por otra parte el descompresor software implementado en CPU Guernika es previo a la realización de este proyecto y lo único que se ha hecho es realizar pruebas sobre su velocidad de descompresión. En estas pruebas sólo se evalúa un campo, este descompresor es ejecutado en un ordenador con las siguientes características:

- Intel Core i7, 2,97 GHz.
- S.O. Debian Squeeze.
- Bogomips: 5.866 instrucciones/seg.
- Memoria RAM: 4 GBytes.
- Memoria caché: 8 Mbytes.

Los resultados obtenidos en los tres casos diferentes son los siguientes:

16 datos repetidos 100 millones de veces - 5.96 GB						
Tipo de datos	**VHDL**		**C en Microblaze**		**C en CPU Guernika**	
	Tiempo (seg)	**Rend (MB/s)**	**Tiempo (seg)**	**Rend (MB/s)**	**Tiempo (seg)**	**Rend (MB/s)**
Diferentes	80	76,29	573	10,65	8,2	744,33
Con una repetición	92,7	65,84	349	17,49	6,6	924,78
Iguales	57,4	106,33	308	19,82	3,57	1709,67

Tabla 29: Comparativa de velocidad de descompresión hardware y software.

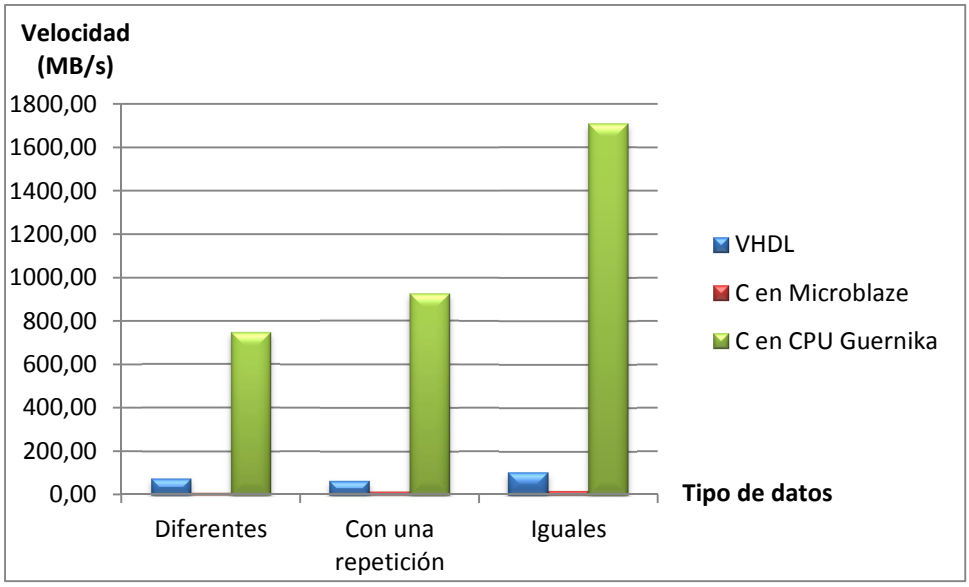


Figura 48: Comparativa de velocidad de descompresión hardware y software.

Para una mejor visualización se va a proceder a mostrar con mejor detalle la comparativa entre VHDL y C en Microblaze:

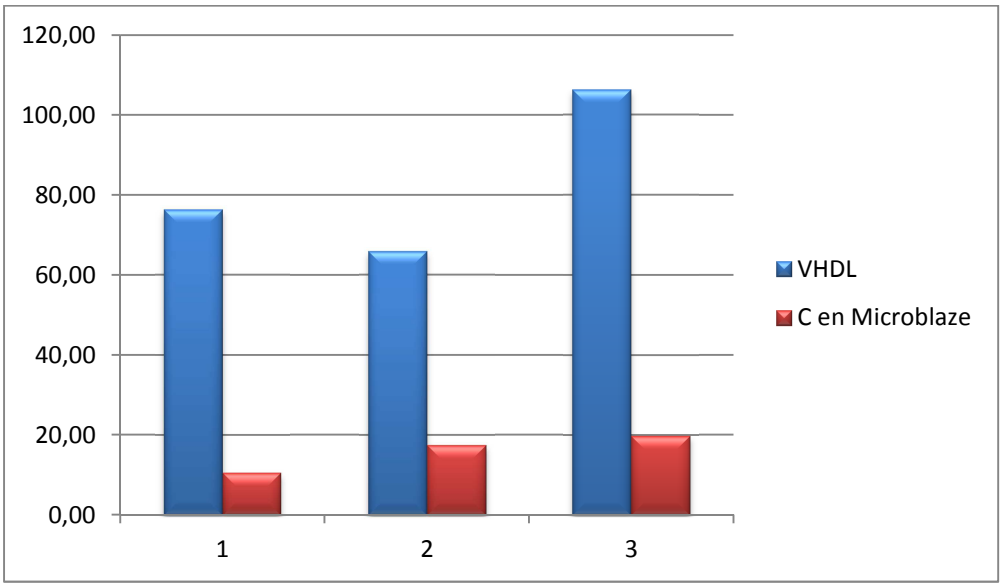


Figura 49: Comparativa de velocidad de descompresión hardware y software en FPGA.

A primera vista en la Figura 48 destaca la notable diferencia entre la arquitectura C implementada en CPU Guernika que en las otras dos, dado que es en torno a 16 veces más veloz que la arquitectura hardware y 76 veces más rápida que la misma arquitectura implementada en Microblaze.

Por otro lado dado que la versión hardware y la implementación software al ejecutarse en la placa disponen de los mismos recursos, por lo tanto, es interesante comparar sus valores, dado que el código VHDL alcanza una velocidad que es hasta 5 veces mayor que la implementación software.

Una vez analizados los datos cabe destacar que dicha diferencia no es tan aparatosa como a primera vista parece, dado que CPU Guernika tiene un procesador que funciona a 2.97 GHz, mientras que tanto el diseño hardware como el software implementado en Microblaze tienen una frecuencia de 50 MHz, es decir **la velocidad del CPU es en torno a 60 veces mayor**, por lo que visto así, en proporción el diseño hardware es el que obtiene un mayor rendimiento, seguido del software en CPU Guernika y en último lugar el software en Microblaze.

El rendimiento de la CPU es mayor que la de Microblaze ya que la frecuencia de trabajo no es el único factor que influye en grandes tramas de datos, también está la memoria RAM y caché, aunque influyen en menor manera. Es decir, aunque la frecuencia de trabajo es 60 veces mayor en la CPU, la velocidad de descompresión es en torno a 76 veces mayor.

Por otro lado comparando el diseño hardware con el de la CPU, la frecuencia es 60 veces mayor pero la velocidad de descompresión sólo es 16 veces mayor, es decir, **a igualdad de recursos el diseño hardware será mucho más rápido** ya que es más eficiente que la arquitectura software. La mayor diferencia reside en que la arquitectura hardware puede empezar la segunda descompresión mientras que la primera aún se está llevando a cabo, mientras que el diseño software es secuencial, es decir tiene que acabar toda la primera descompresión para comenzar con la segunda. Por otra parte en caso de utilizar más campos, la implementación hardware sería más competitiva ya que procesaría en paralelo cada campo.

Para hacer una buena comparación es interesante resaltar el número de ciclos de reloj que se necesitan para procesar un dato, sabiendo la cantidad de datos que se procesan por segundo y la frecuencia a la que trabaja cada sistema se obtiene:

16 datos repetidos 100 millones de veces - 5.96 GB						
Tipo de datos	VHDL (50 MHz)		C en Microblaze (50 MHz)		C en CPU Guernika (2.97 GHz)	
	Rend (MB/s)	Tiempo (ciclos/dato)	Rend (MB/s)	Tiempo (ciclos/dato)	Rend (MB/s)	Tiempo (ciclos/dato)
Diferentes	76,29	0,66	10,65	4,69	744,33	3,99
Con una repetición	65,84	0,76	17,49	2,86	924,78	3,21
Iguales	106,33	0,47	19,82	2,52	1649,60	1,80

Tabla 30: Comparativa ciclos de reloj necesarios por dato en plataforma hardware y software.

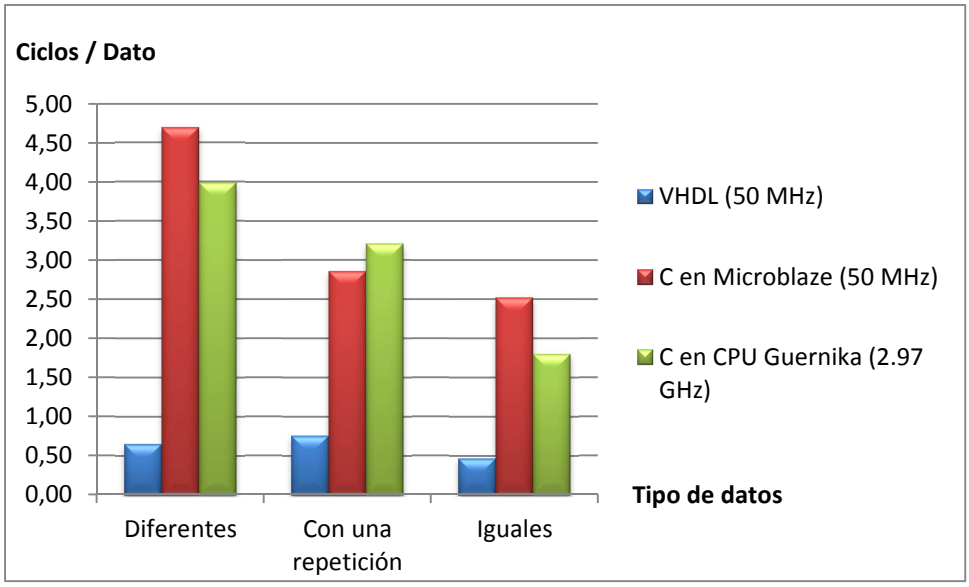


Figura 50: Comparativa ciclos de reloj necesarios por dato en plataforma hardware y software.

Como se observa en la Figura 50, cuando se trabaja con el software se obtiene un rendimiento similar ya que CPU Guernika es notablemente más rápido pero porque trabaja a la misma frecuencia, pero para procesar un dato necesita un número similar de ciclos de reloj que Microblaze. En cambio la versión hardware necesita menos de un ciclo de reloj por dato, lo que significa que a igualdad de recursos es 8 veces más rápido que CPU Guernika. Este valor de ciclos de reloj es menor que uno dado que al ser un elemento hardware los diferentes procesos trabajan en paralelo.

También es interesante, como en casos anteriores comprobar el ratio entre velocidad de procesado y consumo energético:

Rendimiento hardware y software									
Tipo de datos	VHDL (50 MHz)			C en Microblaze (50 MHz)			C en CPU Guernika (2.97 GHz)		
	Rend (MB/s)	Cons (W)	Rend ((MB/s)/W)	Rend (MB/s)	Cons (W)	Rend ((MB/s)/W)	Rend (MB/s)	Cons (W)	Rend ((MB/s)/W)
Diferentes	76,29	1,35	56,51	10,65	1,4	7,61	744,33	55.5	13.42
Una repetición	65,84	1,35	48,77	17,49	1,4	12,49	924,78	55.5	16.67
Iguales	106,33	1,35	78,77	19,82	1,4	14,15	1649,60	55.5	29.74

Tabla 31: Comparativa de datos procesados por vatio entre implementación hardware y software.

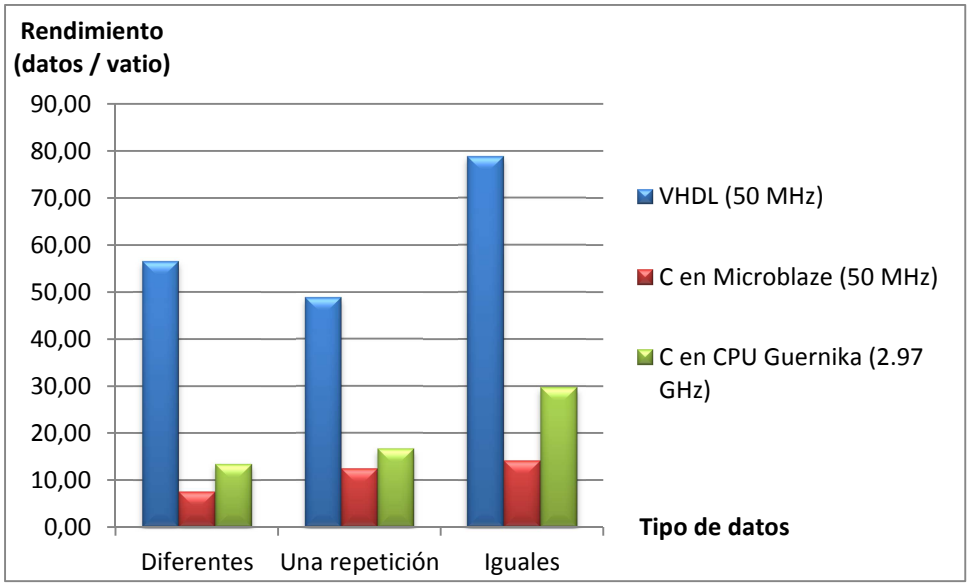


Figura 51: Comparativa de datos procesados por vatio entre implementación hardware y software.

Para realizar el cálculo de potencia consumida en la CPU se ha considerado que la potencia consumida es un 85% de la potencia máxima del procesador y que dicho procesador dispone de 4 núcleos, por lo que la potencia consumida por un núcleo será una cuarta parte de la consumida por la CPU a plena carga.

Como se ha dicho anteriormente, la velocidad de la CPU es muy superior al resto pero porque su frecuencia de trabajo es mucho superior. Se ha visto en la Figura 50 que los ciclos que se necesitan por cada dato, el diseño hardware es superior. En este caso en la Figura 51 se puede observar que si también se tiene en cuenta el consumo, aunque la CPU sea mucho más rápida es el caso en que peor ratio existe entre velocidad y consumo, mientras que en este caso el diseño hardware cobra fuerza y es donde más diferencia existe ya que necesitando menos ciclos por cada dato procesado, también consume menos energía por lo que es la opción más eficiente ya que la implementación software en ambos casos se obtiene resultados similares pero el rendimiento de datos por vatio es en torno a 8 veces superior en el caso del hardware.

En resumen, si el máximo objetivo es la velocidad de descompresión la mejor elección será implementarlo en CPU Guernika. En caso que los requisitos no sean tan altos, el diseño hardware implementado en FPGAs empieza a cobrar fuerza ya que obtiene velocidades rápidas con muchos menos recursos, ya que no hay que tener un ordenador exclusivamente para descomprimir, y un consumo energético muy inferior. Por otra parte se puede implementar el algoritmo en FPGAs más potentes con una frecuencia mayor, por ejemplo la placa Atlys que trabaja a 100 MHz con lo que la velocidad de CPU Guernika disminuirá a 8 veces la del hardware. Hay que tener en cuenta que existen FPGAs que trabajan a mayor frecuencia y el consumo siempre será menor que una CPU, por lo que es una buena opción a la hora de considerar implementar el algoritmo de compresión – descompresión.

6. CONCLUSIONES Y TRABAJOS FUTUROS

En este capítulo se van a enumerar y describir las conclusiones obtenidas después de la implementación del algoritmo en una FPGA y de las pruebas realizadas a los distintos diseños y distintas FPGAS durante la realización de este proyecto fin de carrera. Para acabar se presentará una relación de posibles trabajos futuros para continuar con esta rama dentro del departamento.

6.1. CONCLUSIONES

La principal conclusión que se puede obtener al término de este proyecto de fin de carrera es que el **objetivo principal** de implementar un diseño de descompresión de datos de tamaño de palabras arbitrario en lenguaje VHDL, para la complementación de un diseño de compresión ya existente **se ha completado con éxito**.

El primer obstáculo encontrado ha sido la familiarización con el entorno de desarrollo, XPS, en el que se ha realizado la implementación del algoritmo, ya que no se partía con conocimientos previos del programa. Pero con el uso continuado del programa XPS ha permitido **adquirir los conocimientos necesarios de una forma rápida** lo que ha permitido que el proyecto no se quedase estancado. El principal problema del programa es el tiempo de compilación ya que al principio del proyecto se partía de un equipo antiguo y el tiempo de síntesis era de aproximadamente 40 minutos, más adelante se contó con un equipo nuevo y ese tiempo se redujo a 10 minutos, pero aun así sigue siendo un tiempo elevado ya que la mínima modificación que se haga en el código VHDL conlleva volver a realizar la síntesis y en el proceso de diseño de la arquitectura esto ocurre múltiples veces. Otro problema es que la simulación del programa es compleja lo cual dificulta su uso.

Por estos motivos se ha utilizado el programa Quartus II para tener un apoyo al comienzo del proyecto y poder empezar a desarrollar los primeros diseños. También se disponía de conocimientos previos de este programa lo que permitió empezar con el diseño del algoritmo prácticamente desde el primer día. Por otro lado el programa dispone de una rápida compilación y el sencillo simulador del que este programa consta. Todo ello ha permitido que el desarrollo del proyecto haya sido rápido. Hay que destacar que **no habría sido necesario el uso del Quartus II**, ya que se podrían haber usado los bancos de prueba para las simulaciones, pero sí que hay que recalcar que su uso ha permitido que el proyecto se desarrolle de una forma mucho más dinámica y sencilla.

Por otra parte otro gran problema fue la comunicación FSL, dado que al empezar el proyecto, sólo se tenían conocimientos de como configurar un único FSL de salida de Microblaze, y a priori no era posible aumentar a más FSL, de hecho en el anterior proyecto no pudieron hacerlo, por lo que hubo que investigar como poder añadir más FSL de comunicación ya que para el descompresor era vital tener más de un FSL de salida de Microblaze. Esto llevo tiempo pero finalmente se consiguió, y dado que hay que seguir numerosos pasos y es fácil equivocarse, **se ha realizado un apéndice que consta de un tutorial** con los pasos a seguir para añadir varios FSL de salida de Microblaze.

Como ya se ha comentado en la memoria en primer lugar se desarrolló un algoritmo de descompresión simple. Una vez éste era totalmente funcional se pasó al diseño de la descompresión doble añadiendo un segundo coprocesador. Cuando la descompresión doble fue totalmente operativa se repitieron los pasos anteriores para obtener una descompresión doble de tres campos distintos. Después de esto se verificó que el conjunto funcionaba correctamente y se realizaron las pruebas que se han mostrado en la memoria. Para terminar,

se englobó en un único diseño el compresor y descompresor para validar el funcionamiento del proyecto anterior y del actual, lo cual sucedió de manera exitosa. En este caso también se realizaron pruebas de validación en diferentes dispositivos, que resultaron ser exitosas.

Se ha evaluado la velocidad de la compresión – descompresión en distintos dispositivos, comprobando con que tipos de datos el algoritmo funciona mejor y comparando las distintas configuraciones posibles de los dispositivos.

Se ha evaluado el consumo energético de la compresión – descompresión en distintos dispositivos comprobando la variación de energía consumida en las distintas configuraciones posibles y entre los distintos dispositivos, permitiendo extraer conclusiones de qué FPGA obtiene un mayor rendimiento energético.

También se ha evaluado el coste hardware del algoritmo. Dependiendo de cada dispositivo, se ha comparado la cantidad de recursos hardware que necesita para implementar el diseño y se han comparado entre sí.

A parte del algoritmo diseñado en VHDL, también se ha diseñado un código en lenguaje C, encargado de controlar el funcionamiento de Microblaze. Esto fue necesario para controlar los datos que se enviaban al descompresor y por tanto validar su funcionamiento mostrando los resultados por pantalla.

Por otro lado se ha realizado una comparación entre varios métodos de compresión y descompresión (hardware y software) y se ha evaluado su rendimiento. Se considera que mediante hardware se realiza el algoritmo de forma **rápida y eficiente** ya que a igualdad de recursos la velocidad es mayor que en ninguna otra plataforma con consumos que se pueden considerar bastante bajos teniendo en cuenta que se trata de un dispositivo mucho menos potente (FPGA frente a un PC). Pero hay que tener en cuenta que un PC es mucho más potente que una FPGA por lo que, si lo que prima es la velocidad, CPU Guernika es la opción más idónea.

Personalmente me ha resultado un proyecto interesante ya que he podido aplicar los conocimientos adquiridos durante la carrera de forma práctica, pudiendo ver los resultados obtenidos. Estos conocimientos me han permitido no tener que empezar el proyecto de cero y tener que documentarme para aprender las técnicas del lenguaje VHDL, lo cual ha ayudado a que el proyecto empezara de forma dinámica. Por otra parte también han incrementado mis conocimientos del lenguaje de programación C, que aunque no es la base de este proyecto, ha sido necesario su uso para controlar la FPGA. También es importante la agilidad adquirida con el programa XPS ya que en el mundo laboral, es un programa utilizado para la programación de FPGAs y creo que estos conocimientos prácticos pueden ser muy útiles en el futuro.

El mayor logro del proyecto ha sido comprobar que, lo realizado en el proyecto previo al mío y la complementación que yo he realizado, **funciona correctamente**, lo que quiere decir que ambos hemos hecho nuestro trabajo de forma exitosa y es una gran satisfacción personal.

6.2. TRABAJOS FUTUROS

En este apartado se van a enumerar una serie de posibles trabajos futuros, ya sea de mejora de los proyectos realizados o ampliación de los mismos. Estos trabajos futuros no se han realizado por falta de tiempo, ya que sería hacer otro proyecto de fin de carrera, o porque no entraban en el plan inicial y en el transcurso del proyecto han surgido y pueden ser interesantes. Cabe destacar que según iba avanzando este proyecto se han ido incluyendo las pruebas más interesantes y significativas que en un principio no formaban parte del proyecto, pero hay otras que no se han realizado. Este proyecto que se ha realizado ya parte como un trabajo futuro de otro anterior por lo que es interesante continuar con esta rama en el departamento. A continuación se muestran las propuestas:

1. Integrar el diseño en Petalinux para enviar trazas reales de datos y medir el tiempo de forma más precisa, de esta forma se podrían obtener datos más exactos respecto el rendimiento tanto de las FPGAs como del algoritmo.

   Observar el rendimiento del algoritmo en un sistema operativo (Petalinux) y compararlo con el rendimiento en FPGAs.

2. Hacer un estudio del rendimiento de las FPGAs a distintas temperaturas. Estos elementos son sensibles a la temperatura (como cualquier circuito hardware) y podría resultar interesante los límites de temperatura en la que el diseño funciona correctamente y comprobar si existe alguna temperatura óptima en la que el rendimiento sea mayor.

3. Integrar todo el compresor y el descompresor en un único coprocesador y comparar velocidad de ejecución (ya que no habría que transmitir los datos entre coprocesadores), coste hardware y consumo energético.

   Se ha realizado un diseño con varios coprocesadores para aprovechar el compresor realizado con anterioridad y para hacer un diseño modular.

4. Implementar el mismo compresor – descompresor con otro sistema de comunicación, en vez del FSL (Bus de comunicación AXI) para poder tener tres campos distintos (como en el descompresor) y poder procesar datos en coma flotante, ya que no ha sido posible por la limitación de número de FSL de salida y entrada de Microblaze.

   El bus de comunicación Axi4 Stream es más moderno que el FSL por lo que permitirá también realizar la comunicación de forma más veloz.

5. Comparar el algoritmo de compresión implementado en este proyecto con otros algoritmos de compresión existentes como BZIP2 y GZIP.

   Implementar los algoritmos de compresión en un sistema operativo (como Petalinux) y hacer una comparación de velocidad de compresión entre dichos algoritmos.

7. PRESUPUESTO

En este presupuesto se va a hacer una estimación de los gastos que ha supuesto la realización de este proyecto. Se puede dividir en varias categorías, coste de personal, coste hardware y coste software.

Este proyecto ha tenido una duración aproximada de 5 meses. La fecha de comienzo es finales de Octubre de 2011 y la fecha de finalización a finales de Febrero de 2012. Por lo tanto se va a comenzar con el desglose del presupuesto, comenzando por el coste de personal.

❖ COSTES DE PERSONAL

Este proyecto ha contado con la participación de tres ingenieros, uno junior (el alumno) y dos sénior (los tutores) los 5 meses de duración del proyecto.

El número de horas que ha trabajado el ingeniero junior asciende a 650, y se ha asignado un salario de 16 €/hora.

Por otra parte los dos ingenieros sénior han trabajado durante un total de 200 horas, repartidas entre los dos, es decir, 100 horas cada uno. El saldo que se ha asignado a dichos ingenieros es de 25 €/hora. Por lo tanto el coste de personal de este proyecto ha sido el siguiente:

Concepto	Salario/hora	Horas	Total
Ingeniero sénior	25,00 €	200	5.000,00 €
Ingeniero junior	16,00 €	650	10.400,00 €
		TOTAL	**15.400,00 €**

Tabla 32: Costes de personal.

❖ COSTES DE HARDWARE

En este apartado hay que incluir todos los elementos hardware que se han utilizado en el proyecto, desde el puesto de trabajo, todas las FPGAs utilizadas y el vatímetro para medir la potencia consumida por las FPGAs, todos los precios incluyen el correspondiente IVA.

Concepto	Precio
Puesto de trabajo	589,00 €
Placa Spartan 3A Starter Kit	144,16 €
Placa Nexys 2	113,66 €
Placa Nexys 3	151,79 €
Placa Atlys	266,26 €
Vatímetro Watts up? .Net	180,01 €
TOTAL	**1.444,88 €**

Tabla 33: Costes de hardware.

Con puesto de trabajo se hace referencia a un ordenador con monitor, teclado, ratón y sistema operativo Windows 7 Professional 64 bits. Cabe destacar que las placas utilizadas tienen un coste menor si es para uso académico pero se ha querido hacer un presupuesto como si el proyecto se hubiera desarrollado en una empresa real.

❖ COSTES DE SOFTWARE

En este apartado hay que incluir todas las licencias de los distintos programas utilizados para el desarrollo y la realización de este proyecto. Dichos programas son los siguientes, **Quartus II**, Xilinx Platform Studio que está incluido en el pack **Xilinx ISE Design Suite** y por último **Microsoft Office Professional 2010**.

Concepto	Precio
Quartus II	2.995,00 €
Xilinx ISE Design Suite	2.861,41 €
Microsoft Office Professional 2010	699,90 €
TOTAL	**6.556,31 €**

Tabla 34: Costes de software.

Hay que indicar que las licencias tanto del Quartus II como del Xilinx ISE Design Suite permite la instalación del programa en todos los equipos que se requiera, por lo tanto el precio podría dividirse entre el número total de ordenadores que posean ese programa. El pack Microsoft Office Professional 2010 incluye una única licencia pero permite instalarlo en dos equipos, es decir el precio por equipo es la mitad. Hay que indicar que todos los programas utilizados en este proyecto al ser con fines académicos se han utilizado con licencias gratuitas.

❖ PRESUPUESTO TOTAL

En este apartado se resumen los costes totales y se añade un 7% dedicado al riesgo y se establece un 15% de beneficio.

Concepto	Precio
Costes de personal	15.400,00 €
Costes de hardware	1.444,88 €
Costes de software	6.556,31 €
Subtotal	23.401,19 €
Riesgo (7%)	1.638,08 €
Beneficio (15%)	3.510,18 €
TOTAL PROYECTO	**28.549,45 €**

Tabla 35: Presupuesto total.

El presupuesto total de ejecución de este proyecto asciende a la cantidad de **VEINTIOCHO MIL QUINIENTOS CUARENTA Y NUEVE EUROS CON CUARENTA Y CINCO CÉNTIMOS incluyendo el impuesto sobre el valor añadido.**

28.549.45 € (IVA incluido)

APÉNDICE A. PASOS EN LA IMPLEMENTACIÓN DE UN PROYECTO EN UNA FPGA CON VARIOS FSL

A continuación se va a proceder a la explicación de cómo añadir un bus de comunicación FSL de escritura a un procesador Microblaze.

Se parte de un diseño base ya creado en el que se tienen dos módulos, un procesador Microblaze y un periférico creado por el usuario (llamado "Desc_primario_1"). Este diseño esta comunicado por una par de FSL un master y un esclavo. El objetivo es tener dos FSL master y un esclavo.

Por lo tanto nuestro diseño base será el siguiente:

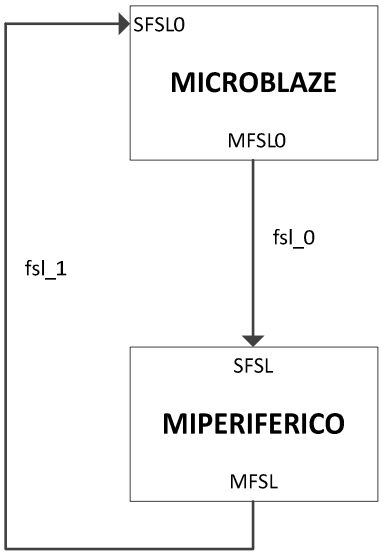


Figura 52: Diseño base.

Y el diseño al que queremos llegar es el siguiente:

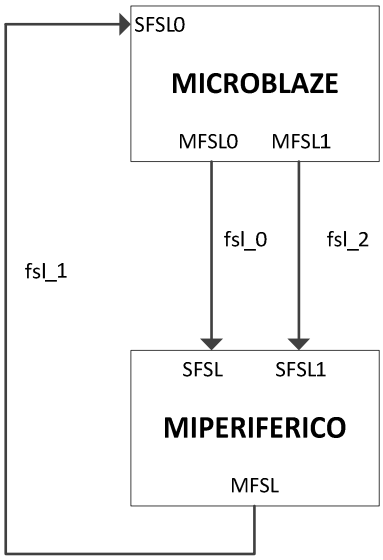


Figura 53: Diseño objetivo.

A continuación se mostrarán los pasos a seguir, para saber dónde están los archivos que hay que modificar tomaré como referencia la carpeta "Proyecto" que es la carpeta en la que se ha creado el diseño en XPS. Para conseguir ese diseño objetivo hay que seguir los siguientes pasos:

1. Editar system.mhs:

Este archivo se encuentra en la carpeta: Proyecto.

En primer lugar hay que configurar las salidas del procesador Microblaze, tenemos que añadir un master MFSL1:

```
48
49  BEGIN microblaze
50   PARAMETER INSTANCE = microblaze_0
51   PARAMETER C_AREA_OPTIMIZED = 1
52   PARAMETER C_USE_BARREL = 1
53   PARAMETER C_DEBUG_ENABLED = 1
54   PARAMETER HW_VER = 8.00.b
55   PARAMETER C_FSL_LINKS = 2
56   BUS_INTERFACE DLMB = dlmb
57   BUS_INTERFACE ILMB = ilmb
58   BUS_INTERFACE DPLB = mb_plb
59   BUS_INTERFACE IPLB = mb_plb
60   BUS_INTERFACE DEBUG = microblaze_0_mdm_bus
61   BUS_INTERFACE SFSL0 = fsl_1
62   BUS_INTERFACE MFSL0 = fsl_0
63   BUS_INTERFACE MFSL1 = fsl_2
64   PORT MB_RESET = mb_reset
65  END
66
```

Figura 54: Modificación system.mhs Microblaze.

A continuación tenemos que configurar las entradas del periférico, tenemos que añadir un esclavo SFSL1, a la vez añadiremos un bus de comunicación FSL:

```
326  BEGIN fsl_v20
327   PARAMETER INSTANCE = fsl_0
328   PARAMETER HW_VER = 2.11.c
329   PARAMETER C_EXT_RESET_HIGH = 1
330   PORT FSL_Clk = clk_50_0000MHz
331   PORT SYS_Rst = sys_bus_reset
332  END
333
334  BEGIN miperiferico
335   PARAMETER INSTANCE = miperiferico_0
336   PARAMETER HW_VER = 1.00.a
337   BUS_INTERFACE MFSL = fsl_1
338   BUS_INTERFACE SFSL = fsl_0
339   BUS_INTERFACE SFSL1 = fsl_2
340   PORT FSL_Clk = clk_50_0000MHz
341  END
342
343  BEGIN fsl_v20
344   PARAMETER INSTANCE = fsl_1
345   PARAMETER HW_VER = 2.11.c
346   PARAMETER C_EXT_RESET_HIGH = 1
347   PORT FSL_Clk = clk_50_0000MHz
348   PORT SYS_Rst = sys_bus_reset
349  END
350
351  BEGIN fsl_v20
352   PARAMETER INSTANCE = fsl_2
353   PARAMETER HW_VER = 2.11.c
354   PARAMETER C_EXT_RESET_HIGH = 1
355   PORT FSL_Clk = clk_50_0000MHz
356   PORT SYS_Rst = sys_bus_reset
357  END
358
```

Figura 55: Modificación system.mhs Desc_primario_1 y creación de FSL.

2. Editar Desc_primario_1_v2_1_0.mpd:

Este archivo se encuentra en la carpeta: Proyecto/pcores/Desc_primario_1_v2_1_0/data.

En segundo lugar hay que conectar la entrada del periférico anteriormente creada (SFSL1) con los puertos de nuestro diseño:

```
## Bus Interfaces
BUS_INTERFACE BUS=SFSL, BUS_STD=FSL, BUS_TYPE=SLAVE
BUS_INTERFACE BUS=SFSL1, BUS_STD=FSL, BUS_TYPE=SLAVE
BUS_INTERFACE BUS=MFSL, BUS_STD=FSL, BUS_TYPE=MASTER

## Peripheral ports
PORT FSL_Clk = "", DIR=I, SIGIS=Clk, BUS=MFSL:SFSL:SFSL1
PORT FSL_Rst = OPB_Rst, DIR=I, BUS=MFSL:SFSL:SFSL1
PORT FSL_S_Clk = FSL_S_Clk, DIR=I, SIGIS=Clk, BUS=SFSL
PORT FSL_S_Read = FSL_S_Read, DIR=O, BUS=SFSL
PORT FSL_S_Data = FSL_S_Data, DIR=I, VEC=[0:31], BUS=SFSL
PORT FSL_S_Control = FSL_S_Control, DIR=I, BUS=SFSL
PORT FSL_S_Exists = FSL_S_Exists, DIR=I, BUS=SFSL
PORT FSL1_S_Clk = FSL_S_Clk, DIR=I, SIGIS=Clk, BUS=SFSL1
PORT FSL1_S_Read = FSL_S_Read, DIR=O, BUS=SFSL1
PORT FSL1_S_Data = FSL_S_Data, DIR=I, VEC=[0:31], BUS=SFSL1
PORT FSL1_S_Control = FSL_S_Control, DIR=I, BUS=SFSL1
PORT FSL1_S_Exists = FSL_S_Exists, DIR=I, BUS=SFSL1
PORT FSL_M_Clk = FSL_M_Clk, DIR=I, SIGIS=Clk, BUS=MFSL
PORT FSL_M_Write = FSL_M_Write, DIR=O, BUS=MFSL
PORT FSL_M_Data = FSL_M_Data, DIR=O, VEC=[0:31], BUS=MFSL
PORT FSL_M_Control = FSL_M_Control, DIR=O, BUS=MFSL
PORT FSL_M_Full = FSL_M_Full, DIR=I, BUS=MFSL

END
```

Figura 56: Modificación Desc_primario_1_v2_1_0.mpd.

3. Editar Desc_primario_1.vhd:

Este archivo se encuentra en la carpeta: Proyecto/pcores/Desc_primario_1_v2_1_0/hdl/vhdl.

En tercer lugar hay que añadir a nuestro diseño en VHDL los nuevos puertos:

```
entity miperiferico is
    port
    (
        -- DO NOT EDIT BELOW THIS LINE ----------------------
        -- Bus protocol ports, do not add or delete.
        FSL_Clk : in    std_logic;
        FSL_Rst : in    std_logic;

        FSL_S_Clk   : in    std_logic;
        FSL_S_Read  : out   std_logic;
        FSL_S_Data  : in    std_logic_vector(0 to 31);
        FSL_S_Control   : in    std_logic;
        FSL_S_Exists    : in    std_logic;

        FSL1_S_Clk  : in    std_logic;
        FSL1_S_Read : out   std_logic;
        FSL1_S_Data : in    std_logic_vector(0 to 31);
        FSL1_S_Control  : in    std_logic;
        FSL1_S_Exists   : in    std_logic;

        FSL_M_Clk   : in    std_logic;
        FSL_M_Write : out   std_logic;
        FSL_M_Data  : out   std_logic_vector(0 to 31);
        FSL_M_Control   : out   std_logic;
        FSL_M_Full  : in    std_logic
        -- DO NOT EDIT ABOVE THIS LINE ----------------------
    );

attribute SIGIS : string;
attribute SIGIS of FSL_Clk : signal is "Clk";
attribute SIGIS of FSL_S_Clk : signal is "Clk";
attribute SIGIS of FSL1_S_Clk : signal is "Clk";
attribute SIGIS of FSL_M_Clk : signal is "Clk";

end miperiferico;
```

Figura 57: Modificación Desc_primario_1.vhd.

4. Editar Desc_primario_1_v2_1_0_app.c:

Este archivo se encuentra en la carpeta: Proyecto/drivers/Desc_primario_1_v1_00_a/examples.

En cuarto lugar hay que modificar el código C para poder hacer la simulación del diseño:

```
30  #define WRITE_MIPERIFERICO_0(val)   write_into_fsl(val, XPAR_FSL_MIPERIFERICO_0_INPUT_SLOT_ID)
31  #define WRITE_MIPERIFERICO_1(val)   write_into_fsl(val, XPAR_FSL_MIPERIFERICO_1_INPUT_SLOT_ID)
32  #define READ_MIPERIFERICO_0(val)    read_from_fsl(val, XPAR_FSL_MIPERIFERICO_0_OUTPUT_SLOT_ID)
33
34  void miperiferico_app(
35        unsigned int* input_0,       /* Array size = 1 */
36        unsigned int* input_1,       /* Array size = 1 */
37        unsigned int* output_0       /* Array size = 1 */
38        )
39  {
40     int i;
41
42     //Start writing into the FSL bus
43     for (i=0; i<1; i++)
44     {
45       WRITE_MIPERIFERICO_0(input_0[i]);
46       WRITE_MIPERIFERICO_1(input_1[i]);
47     }
48
49     //Start reading from the FSL bus
50     for (i=0; i<1; i++)
51     {
52       READ_MIPERIFERICO_0(output_0[i]);
53     }
54  }
55
56  main()
57  {
58       unsigned int input_0[1];
59       unsigned int input_1[1];
60       unsigned int output_0[1];
61
62
63  #ifdef __PPC__
64       // Enable APU for PowerPC.
65       unsigned int msr_i;
66       msr_i = mfmsr();
67       msr_i = (msr_i | XREG_MSR_APU_AVAILABLE | XREG_MSR_APU_ENABLE) & ~XREG_MSR_USER_MODE;
68       mtmsr(msr_i);
69  #endif
70
71       //Initialize your input data over here:
72       input_0[0] = 12345;
73       input_1[0] = 12345;
```

Figura 58: Modificación Desc_primario_1_v2_1_0_app.c parte 1.

```
75      //Call the macro with instance specific slot IDs
76      miperiferico(
77          XPAR_FSL_MIPERIFERICO_0_INPUT_SLOT_ID,
78          XPAR_FSL_MIPERIFERICO_1_INPUT_SLOT_ID,
79          XPAR_FSL_MIPERIFERICO_0_OUTPUT_SLOT_ID,
80          input_0,
81          input_1,
82          output_0
83          );
84
85      // You can also define your own function to access the peripheral
86      // Here you are calling the example function defined above
87      // Note the slot ID can not be passed over as function parameters
88      miperiferico_app(
89          input_0,
90          input_1,
91          output_0
92          );
93
94  }
```

Figura 59: Modificación Desc_primario_1_v2_1_0_app.c parte 2.

Antes de seguir con los pasos restantes abriremos nuestro diseño con el XPS. Si hemos tenido algún fallo en uno de los pasos anteriores el proyecto no se abrirá y en la consola inferior aparecerá un error diciéndonos en que paso hemos fallado.

Se puede comprobar en las pestañas inferiores “Block Diagram” y “System Assembly View” que el diseño se ha modificado correctamente.

5. Compilar el diseño:

En quinto lugar se volverá a compilar, ya que hemos modificado el código VHDL. Primero en el menú superior en “Project” se seleccionará “Clean All Generated Files”. Después en el menú superior en “Hardware” se seleccionará “Generate Bitstream”. Una vez haya compilado y no tengamos errores continuaremos con el siguiente paso.

6. Compilar el código C:

En sexto lugar se compilará el código C.

Al compilar dará error y tendremos que modificar los archivos, “Desc_primario_1.h” y “xparameters.h”, pero antes hay que compilar para que cree esos dos archivos.

No modificar “Desc_primario_1.h” que está ubicado en: Proyecto/drivers/Desc_primario_1_v1_00_a/src.

Para compilar, en la ventana izquierda en la pestaña “Applications” se pulsará botón derecho del ratón encima de la aplicación software que hayamos creado al crear el proyecto y se dará a “Build Project”.

El error que dará será el siguiente:

```
drivers/miperiferico_v1_00_a/examples/miperiferico_v2_1_0_app.c:83:4: error: macro "miperiferico" passed 6 arguments, but takes just 4
drivers/miperiferico_v1_00_a/examples/miperiferico_v2_1_0_app.c: In function 'main':
drivers/miperiferico_v1_00_a/examples/miperiferico_v2_1_0_app.c:76: error: 'miperiferico' undeclared (first use in this function)
drivers/miperiferico_v1_00_a/examples/miperiferico_v2_1_0_app.c:76: error: (Each undeclared identifier is reported only once
drivers/miperiferico_v1_00_a/examples/miperiferico_v2_1_0_app.c:76: error: for each function it appears in.)
make: *** [soft_miperiferico/executable.elf] Error 1
Done!
```

Figura 60: Errores primera compilación del código C.

7. Editar Desc_primario_1.h:

Este archivo se encuentra en la carpeta: Proyecto/microblaze_0/include

En séptimo lugar se añadirá un dato de escritura al igual que en el código C:

```
53  #define  miperiferico(\
54           input_slot_id,\
55           input_slot_id1,\
56           output_slot_id,\
57           input_0,      \
58           input_1,      \
59           output_0      \
60           )\
61  {\
62     int i;\
63  \
64     for (i=0; i<1; i++)\
65     {\
66       write_into_fsl(input_0[i], input_slot_id);\
67     }\
68  \
69     for (i=0; i<1; i++)\
70     {\
71       write_into_fsl(input_1[i], input_slot_id1);\
72     }\
73  \
74     for (i=0; i<1; i++)\
75     {\
76       read_from_fsl(output_0[i], output_slot_id);\
77     }\
78  }
79
```

Figura 61: Modificación Desc_primario_1.h

8. Compilar el código C:

En octavo lugar volveremos a compilar el código C al igual que en el paso 6. Volverá a mostrar errores esta vez por el archivo “xparameters.h”.

El error que dará será el siguiente:

```
/cygdrive/c/Users/equipo/AppData/Local/Temp/ccIe4epQ.s: Assembler messages:
/cygdrive/c/Users/equipo/AppData/Local/Temp/ccIe4epQ.s:27: Error: register expected, but saw 'rfslXP'
/cygdrive/c/Users/equipo/AppData/Local/Temp/ccIe4epQ.s:27: Warning: ignoring operands: rfslXPAR_FSL_MIPERIFERICO_1_INPUT_SLOT_ID
/cygdrive/c/Users/equipo/AppData/Local/Temp/ccIe4epQ.s:69: Error: register expected, but saw 'rfslXP'
/cygdrive/c/Users/equipo/AppData/Local/Temp/ccIe4epQ.s:69: Warning: ignoring operands: rfslXPAR_FSL_MIPERIFERICO_1_INPUT_SLOT_ID
make: *** [soft_miperiferico/executable.elf] Error 1
Done!
```

Figura 62: Errores segunda compilación del código C.

9. Editar xparameters.h:

Este archivo se encuentra en la carpeta: Proyecto/microblaze_0/include

En noveno lugar se añadirá la lectura y escritura del nuevo bus de comunicación FSL. Por defecto al crear un nuevo FSL conectado al Microblaze crea un master y un esclavo aunque solo vayamos a usar uno de ellos:

```
157  /******************************************************************/
158
159  #define XPAR_FSL_MIPERIFERICO_0_OUTPUT_SLOT_ID  0
160  #define XPAR_FSL_MIPERIFERICO_0_INPUT_SLOT_ID  0
161  #define XPAR_FSL_MIPERIFERICO_1_OUTPUT_SLOT_ID  1
162  #define XPAR_FSL_MIPERIFERICO_1_INPUT_SLOT_ID  1
163
164  /******************************************************************/
```

Figura 63: Modificación xparameters.h

10. Compilar el código C:

En décimo lugar volveremos a compilar el código C al igual que en pasos anteriores. Si todos los pasos se han seguido correctamente no debería dar ningún error. En caso que haya algún error revisar de qué error se trata y volver a realizar los pasos necesarios.

8. BIBLIOGRAFÍA

[AALG]

Alegsa. Información acerca de las FPGA y sus aplicaciones.

http://www.alegsa.com.ar/Dic/fpga.php

[ALT]

Alt1040. Información acerca de las FPGA y sus aplicaciones.

http://alt1040.com/2010/09/fpga-y-el-sorprendente-poder-del-hardware-reconfigurable

[Digilent-Atlys]

Digilent. Especificaciones FPGA Atlys.

http://www.digilentinc.com/Data/Products/ATLYS/Atlys_rm.pdf

[Digilent-Nexys2]

Digilent. Especificaciones FPGA Nexys 2.

http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf

[Digilent-Nexys3]

Digilent. Especificaciones FPGA Nexys 3.

http://www.digilentinc.com/Data/Products/NEXYS3/Nexys3_rm.pdf

[FL – FPGA]

FPGA libre. Definición y características de FPGA.

http://fpgalibre.sourceforge.net/

[FSL]

Xilinx. Información sobre el bus de comunicación FSL:

http://www.xilinx.com/support/documentation/ip_documentation/fsl_v20.pdf

[Micro-MB]

Microcontroladores PIC. Información acerca del microprocesador Microblaze:

http://www.microcontroladorespic.com/tutoriales/FPGAs/Procesador-MicroBlaze.html

[NI – FPGA]

National Instruments. Definición de una FPGA.

http://zone.ni.com/devzone/cda/tut/p/id/8259

[TH–Filtering]

Think Mind. Diferencia entre escritura síncrona y asíncrona.

http://www.thinkmind.org/download.php?articleid=content_2010_2_30_60017

[Xilinx – Spartan3]

Xilinx. Información sobre la FPGA Spartan-3ª.

http://www.xilinx.com/products/boards-and-kits/HW-SPAR3A-SK-UNI-G.htm

[W - FPGA]

Wikipedia. Definición de FPGA y sus aplicaciones.

http://es.wikipedia.org/wiki/Field_Programmable_Gate_Array

[WUM]

Watts up meters. Características del vatímetro.

https://www.wattsupmeters.com/secure/products.php?pn=0